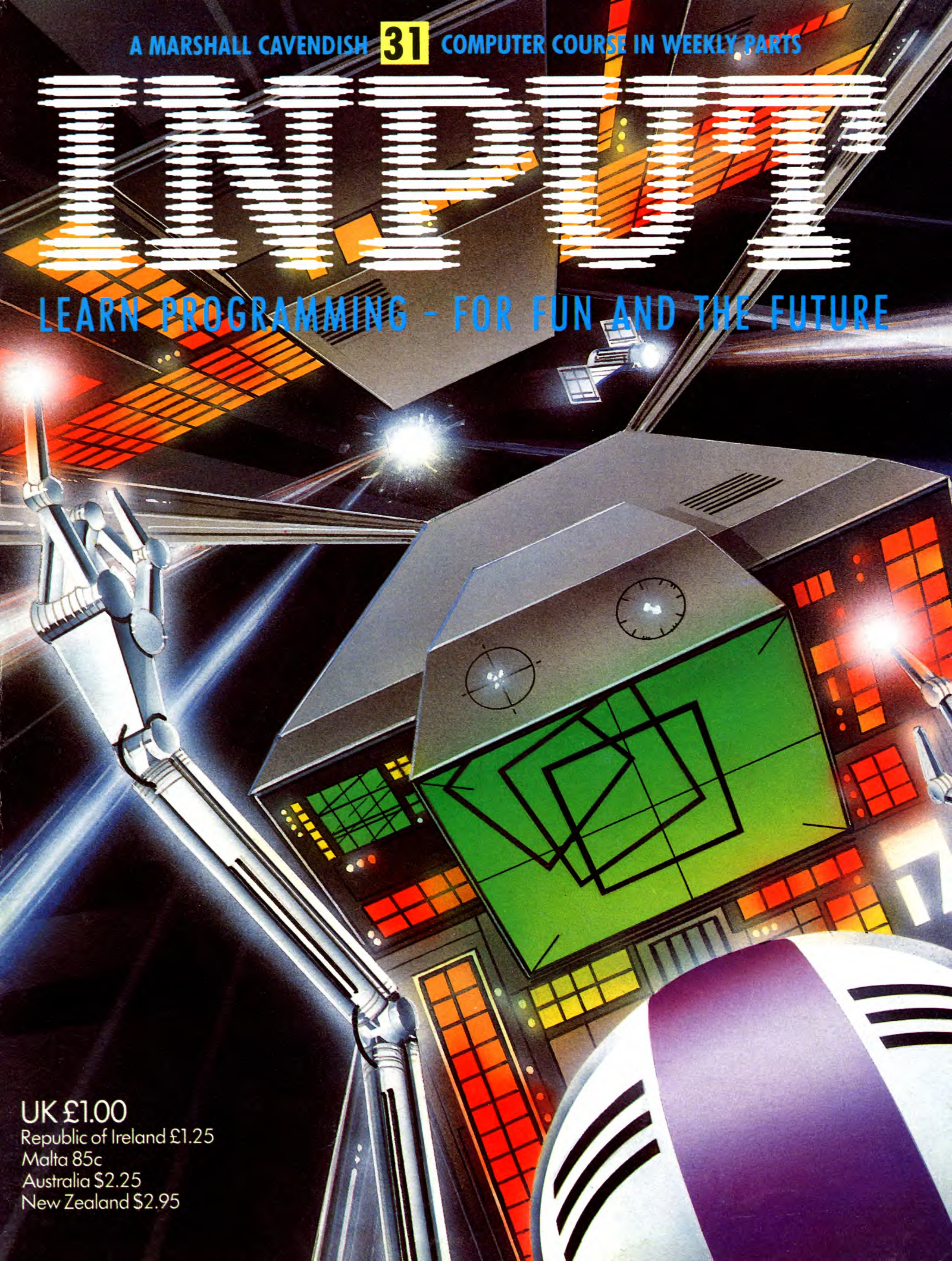

A MARSHALL CAVENDISH 31 COMPUTER COURSE IN WEEKLY PARTS
INPUT
LEARN PROGRAMMING - FOR FUN AND THE FUTURE
UK £1.00
Republic of Ireland £1.25
Malta 85c
Australia $2.25
New Zealand $2.95



Vol. 3 No 31

Contents

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front cover, Ian Stephen. Page 953, Malcolm Harrison. Page 954, Graeme Harris/Chris Lyon. Pages 956, 957, 961, 962, 965, Peter Reilly. Pages 960, 963, 964, Pat Wheelon. Pages 966, 967, 968, 973, Mickey Finn. Pages 974, 979, Ian Stephen. Pages 980, 981, 983, 984, Ellis Nadler.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.



HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland:
Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd,
Department 980, Newtown Road,
Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Times Consultants, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

There are four binders each holding 13 issues.

BACK NUMBERS

Back numbers are supplied at the regular cover price (subject to availability).

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services,
Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent.

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:

Subscription Department, Marshall Cavendish Services Ltd,
Newtown Road, Hove, Sussex BN3 7DN

Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:





COMMODORE 64 and 128





DRAGON 32 and 64



ZX81





EXPLORING THE ACORN'S PAINTBOX

- THE NEW PLOT COMMANDS
- FILLING THE SIMPLE SHAPES
- FINDING THE ENDS OF THE LINE
- FILLING COMPLEX SHAPES
- AN ALL-PURPOSE FILL ROUTINE

Colour graphics commands on the Acorns take a little time to master. However, their sophisticated BASIC does mean that you can easily achieve spectacular results

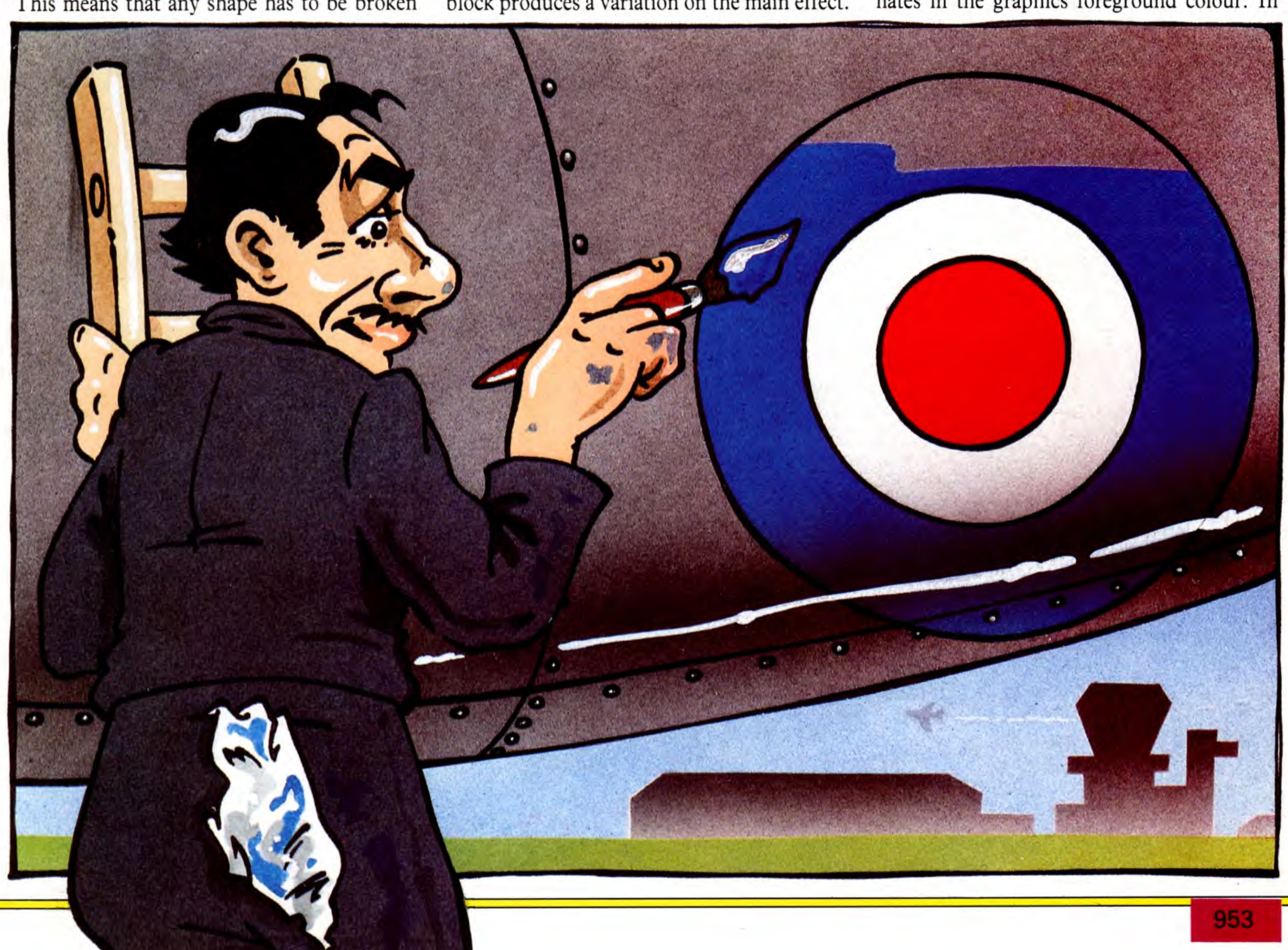
Filling in areas of colour on the Acorns may not be as easy as on some other computers that have a simple PAINT command. But the Acorn commands are a lot more versatile and once you know how to use them, they can be used to colour in any shape you can draw, no matter how complicated.

Early versions of the BBC, with operating system Ø.1, can only fill in triangular shapes using the commands PLOT 8Ø to PLOT 87. This means that any shape has to be broken down into a series of triangles which are filled in one at a time. The triangle-fill commands are quick acting but they are obviously limited to fairly simple shapes. The Electron, and later versions of the BBC, however, have two extra sets of PLOT commands and this article shows how to use these and how versatile they are.

THE NEW PLOT COMMANDS

The Acorns actually have 64 different PLOT commands. These are divided into eight blocks of eight, with each block doing something different such as drawing a line, drawing a dotted line, plotting a dot, filling a triangle, and now the two new sets which fill in a line. Also, each command within the block produces a variation on the main effect. These variations are the same within each block and they are listed in the table overleaf for the series PLOT Ø,x,y to PLOT 7,x,y,. The K stands for the first number after PLOT. You can probably see that the block of eight can be broken down again into two halves. The first four commands treat x and y as relative coordinates measured from the last point visited, while the last four treat x and y as absolute screen coordinates. These distinctions are the same for every block of commands including the two new sets so these are shown in the table as well.

In practice, very few of these variations are used. The sixth is the most common—PLOT 5 in the first series and PLOT 77 and PLOT 93 in the new series. These draw to absolute coordinates in the graphics foreground colour. In



DIFFERENT PLOT EFFECTS

K	x, y	line
Ø 72 88	move relative	don't draw line
1 73 89	move relative	foreground colour
2 74 9Ø	move relative	logical inverse colour
3 75 91	move relative	background colour
4 76 92	move absolute	don't draw line
5 77 93	move absolute	foreground colour
6 78 94	move absolute	logical inverse colour
7 79 95	move absolute	background colour

fact PLOT 5 is so useful it has another name—DRAW. Here is how the new commands work.

FILLING A LINE

First consider the PLOT 72 series. The result of the statement PLOT 77,x,y for instance, is as follows. First, the graphics cursor moves to the point x,y on the screen. Next, it moves horizontally to the left until it finds a point not in the current background colour—usually the outline of the shape you're filling in. Finally, the cursor moves horizontally to the right, drawing a line in the current graphics foreground colour until it again finds a point not in the background colour. The colour of the line is affected by any GCOL statement that may have been used earlier. For example, GCOL 2,1 sets the current foreground colour to number 1 and the current foreground action to AND. This means that every time a point is filled in with the foreground colour, that colour is ANDed with the colour already there (see pages 371 to 373).

If you check back at the table to see how the other PLOT series work, then you will see that the remaining PLOT statements in the series work in a similar way. PLOTs between 72 and 75 use x and y as relative coordinates; PLOTs 72 and 76 don't draw a line, but just move the cursor; PLOTs 74 and 78 invert the colour at each point they draw a line through; and PLOTs 75 and 79 draw the line in the current background colour.

The PLOT 88 to 95 series of statements are similar (if rather less useful) except that they only search horizontally to the right and stop when a point in the background colour is found. There are the usual options for absolute or relative motion and line colour, and these are all given in the table.

FILLING SIMPLE SHAPES

This short program shows how the PLOT 77 command works in practice:

```
1Ø MODE1
2Ø VDU29,64Ø;512;
3Ø PROCTEST
1ØØ END
11Ø DEFPROCTEST
12Ø MOVE−4ØØ,−4ØØ:DRAW−4ØØ,4ØØ
13Ø MOVE4ØØ,−4ØØ:DRAW4ØØ,4ØØ
14Ø PLOT77,Ø,Ø
15Ø ENDPROC
```

Try RUNing the program, and you can see that the effect is to produce a horizontal line across the screen between the two vertical ones. Line 1Ø selects MODE1, Line 2Ø uses VDU29 to put the origin of the graphics coordinates at the centre of the screen. Line 3Ø calls PROCTEST which runs from Lines 11Ø to 14Ø. As you can see, Lines 12Ø and 13Ø simply draw two vertical lines. The interesting part is Line 14Ø, this does a PLOT 77 at the centre of the screen.

If you understand why this program works, then you understand how to use most of the PLOT fill statements. Try to follow what is happening in terms of the description given earlier.

These statements can now be used to do something useful. Change Lines 3Ø to 8Ø of the program in your computer to the following, and add Lines 19Ø to 25Ø:

```
3Ø R%=4ØØ
4Ø GCOLØ,1
5Ø PROCCIRCLE(Ø,Ø,R%,Ø,2*PI)
6Ø FOR YP%=−R% TO R% STEP4
7Ø PLOT77,Ø,YP%
8Ø NEXT
19Ø DEFPROCCIRCLE(XC%,YC%,
   R%,OL,OH)
2ØØ DS=PI/16
21Ø MOVEXC%+R%*COSOL,YC%+
   R%*SINOL
22Ø FOR O=OL TO OH STEP DS
23Ø DRAWXC%+R%*COSO,YC%+
   R%*SINO:NEXT
24Ø DRAWXC%+R%*COSOL,YC%+
   R%*SIN−OL
25Ø ENDPROC
```

Guess what this program does and then RUN it. Clearly, the combination of the PLOT 77 command drawing horizontal lines and the FOR ... NEXT loop moving the PLOT point vertically fills in the circle.

Lines 19Ø to 25Ø are a procedure PROCCIRCLE, which draws an arc of a circle with centre XC%, YC% and radius R%; the starting angle of the arc is OL to the horizontal and the finishing angle is OH. Lines 3Ø and 4Ø select a radius of 4ØØ and set the foreground colour to 1 (red in MODE 1). Line 5Ø uses PROCCIRCLE to draw a circle in the middle of the screen. The actual filling is done by the FOR ... NEXT loop in Lines 6Ø to 8Ø. Inside this loop, a PLOT 77 statement is done for every point with an x coordinate of zero and a y coordinate between the very bottom of the circle and the top.



This is a general technique for filling simple shapes. Just use a PLOT fill along a vertical line from the highest to the lowest point of the shape you want to fill. In the program a STEP of 4 is used in the FOR ... NEXT loop this is because there are 4 vertical graphics units for every point on the screen. By using a STEP of 4, no points are missed, and the program runs faster.

Next change Lines 3Ø to 9Ø of the program to the following:

```
3Ø FORI%=1TO8
4Ø GCOLØ,I%:GCOLØ,128+I%−1
5Ø R%=5ØØ−I%*6Ø
6Ø PROCCIRCLE(Ø,Ø,R%,Ø,2*PI)
7Ø FOR YP%=−R% TO R% STEP 4
8Ø PLOT77,Ø,YP%
9Ø NEXT:NEXT
```

This program draws a series of filled concentric circles; try RUNning it. Lines 3Ø to 9Ø are a FOR ... NEXT loop which steps I% from 1 to 8. Line 4Ø sets the foreground colour to I% and the background colour to I%−1. Note that background colours have 128 added to them; this is how the computer knows that they are background colours. Line 5Ø selects a radius which gets smaller as I% increases. Lastly, Lines 6Ø to 9Ø are the same as the previous example for filling in a circle. The program works by drawing a filled circle in the foreground colour. It then changes the background colour to the foreground colour and changes the foreground colour to the next one. This shows that you can use the usual technique for filling circles even on a coloured background. Now change Line 2ØØ to DS=PI/2 and RUN the program. Try other values for the number in this line; 2, 4, 6, 8 and so on.

ANIMATION

These simple filling techniques can be used to produce animation. DELETE lines 4Ø to 9Ø and add Line 3Ø and DEFPROCEGG (Lines 29Ø to 53Ø):

```
3Ø PROCEGG
29Ø DEFPROCEGG
3ØØ VDU19,3,2,Ø,Ø,Ø,19,1,4,Ø,Ø,Ø
31Ø MOVE−18Ø,−32Ø:MOVE18Ø,−32Ø
32Ø PLOT85,18Ø,32Ø:MOVE−18Ø,32Ø
33Ø PLOT85,−18Ø,−32Ø
34Ø GCOLØ,131:GCOLØ,2
35Ø PROCCIRCLE(Ø,15Ø,15Ø,Ø,2*PI)
36Ø FORYP%=ØTO296:PLOT77,Ø,
  YP%:NEXT
37Ø GCOLØ,1
38Ø PROCCIRCLE(Ø,−15Ø,15Ø,Ø,2*PI)
39Ø FORYP%=−296TOØ:PLOT77,Ø,
  YP%:NEXT
4ØØ FORYP%=−3ØØTOØ
41Ø GCOLØ,13Ø:GCOLØ,1:PLOT77,Ø,−YP%
42Ø GCOLØ,129:GCOLØ,2:PLOT77,Ø,
  YP%:NEXT
43Ø IFINKEY(1ØØ)=9 ENDPROC
44Ø VDU19,1,3,Ø,Ø,Ø,19,2,4,Ø,Ø,Ø
45Ø IF INKEY(1ØØ)=9 ENDPROC
46Ø FORYP%=−3ØØTOØ
47Ø GCOLØ,13Ø:GCOLØ,1:PLOT77,Ø,
  YP%
48Ø GCOLØ,129:GCOLØ,2:PLOT77,Ø,
  −YP%
49Ø NEXT
5ØØ IFINKEY(1ØØ)=9 ENDPROC
51Ø VDU19,2,3,Ø,Ø,Ø,19,1,4,Ø,Ø,Ø
52Ø IFINKEY(1ØØ)=9 ENDPROC ELSE 4ØØ
53Ø ENDPROC
```



This program is a great breakthrough for micro-computing—the high speed egg timer! RUN the program. Actually using it to boil an egg is, however, not recommended. When you tire of the program press the TAB key and it will stop. All the works of this program are contained in PROCEGG. Line 3ØØ uses VDU19 to set colour 3 to green and colour 1 to blue, colour 2 stays as yellow. Lines 31Ø to 33Ø draw the green square. They do this using the PLOT 85 triangle fill statement, by drawing the square as two triangles. Notice how much faster this fill works than the PLOT 77 command. Whenever possible, it is probably best to break up an area to be filled into triangles and use this technique. It is only when filling complex shapes that this is not feasible.

In Line 34Ø the background colour is changed to number 3 (green) and the usual circle fill technique draws a yellow circle (Lines 34Ø to 36Ø) and a blue circle (Lines 37Ø to 39Ø). The clever part of the program is Lines 4ØØ to 42Ø. These are a FOR ... NEXT loop which sweeps YP% from −3ØØ to Ø (the y coordinates of the circles run from −3ØØ to Ø and Ø to 3ØØ). In Line 42Ø the background colour is set to 2, yellow, and the foreground colour to 1, blue. PLOT 77 is then used at Ø,−YP% to produce the blue lines at the top

of the timer. Similarly, Line 41Ø produces the yellow ones at the bottom. Line 45Ø uses the INKEY function to provide a time delay and if the TAB key (number 9) has been pressed it ends the program (Lines 5ØØ and 52Ø do the same). The real point here is to get a time delay when the egg timer starts and finishes. Line 44Ø swops the actual colours corresponding to the colour numbers 1 and 2 (so blue goes to yellow and yellow goes to blue). This is how the timer appears to invert. Lines 46Ø to 49Ø again fill up the inverted timer. At last, Line 51Ø does another colour swop and if TAB has not been pressed, the program loops back to Line 4ØØ.

KEEPING TRACK

To progress beyond these simple techniques, it is necessary to introduce another feature. This allows you to find the coordinates of the ends of the lines drawn by the PLOT fill statements and so fill more complicated shapes. Enter the following new Lines 3Ø to 9Ø and the two procedures PROCSETUP and PROCENDS at Lines 55Ø to 75Ø:

```
30 PROCSETUP
40 PROCTEST
50 PROCENDS
60 PRINTTAB(10,10)"XL% = ";XL%
70 PRINTTAB(22,10)"YL% = ";YL%
80 PRINTTAB(10,12)"XR% = ";XR%
90 PRINTTAB(22,12)"YR% = ";550YR%
   DEFPROCSETUP
560 OSWORD=&FFF1
570 DIM P% 8
580 X%=P% MOD256
590 Y%=P% DIV 256
600 A%=&D
610 ENDPROC
650 DEFPROCENDS
660 CALL OSWORD
670 XL%=P%?0+256*P%?1
680 IFXL%>=&8000 XL%=XL%-&10000
690 YL%=P%?2+256*P%?3
700 IFYL%>=&8000 YL%=YL%-
   &10000
710 XR%=P%?4+256*P%?5
720 IFXR%>=&8000 XR%=XR%-
   &10000
730 YR%=P%?6+256*P%?7
740 IFYR%>=&8000 YR%=YR%-
   &10000
750 ENDPROC
```

In this program, the procedure PROCTEST introduced at the beginning is used again. First though, PROCSETUP is called. This sets things up so that PROCENDS works. You just use PROCSETUP once at the start of every program with PROCENDS in it. When PROCENDS is called, it returns in the variables



Filling shapes a line at a time...

XL%, YL% and XR%, YR%, the coordinates of the last two points visited by the graphics cursor. After a PLOT 77 statement, these are the coordinates of the ends of the line drawn. Incidentally, this technique is general and will return the points that any DRAW and MOVE statements go to. In the program, the ends of the PLOT 77 line are PRINTed out (Lines 6Ø to 9Ø). RUN the program and convince yourself that the values printed are correct.

In MODE 1 there are 4 horizontal graphics units for each screen point so the program gives the x coordinates of the ends of the horizontal line as +396 and −396. Also note that the y coordinates are redundant in this application since they will always be the same as the values used in the PLOT 77 which you had to set up yourself in the first place.

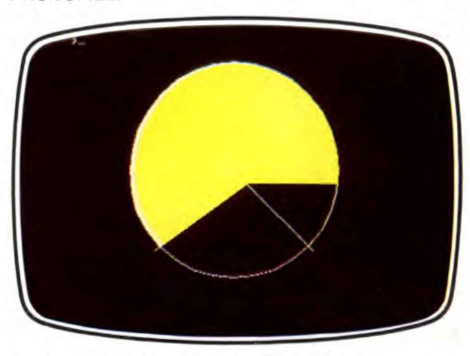
In order actually to find the ends of the lines, it is necessary to use one of the standard subroutines in the operating system; this is the OSWORD routine which will be used to pass back the coordinates. PROCSETUP arranges things for this routine to be called. First the address of OSWORD is put into the variable OSWORD and then a block of memory 8 bytes long at the address P% is reserved using DIM. This will be used by OSWORD to pass back the coordinates using two bytes for each number. In Line 57Ø, the address P% is put into X% and Y%. Finally, A% is set equal to &D, 13 in decimal, the code that tells the routine to return the last two positions of the graphics cursor.

PROCENDS starts by CALLing OSWORD; the rest of the procedure then converts the data which OSWORD has put into the block of memory at P% into coordinates. For instance, Line 67Ø changes two bytes into a two byte number and Line 68Ø turns this from two's complement form into a normal signed BASIC variable.

These last two procedures are best considered as a 'black box'; if you don't understand how they work, don't worry, just make sure that you understand what they do.

FILLING COMPLICATED SHAPES

Once you know how to find the ends of the lines drawn by PLOT 77 you can construct a general purpose fill routine. The idea is that an area is defined by drawing a curve around it. The fill procedure is then called at a single point inside the curve and fills it. Ideally, this should work no matter how complicated the shape. A first attempt at this is given below. As usual modify Lines 4Ø to 6Ø, delete Lines 7Ø to 9Ø and type in the procedure PROCSFILL:



may sometimes cause problems...

```
40 PROCCIRCLE(0,0,400,0,2*PI)
50 GCOL0,2
60 PROCSFILL(0,0)
790 DEFPROCSFILL(XP%,YP%)
800 XF%=XP%:YF%=YP%
810 REPEAT
820 PLOT77,XF%,YF%:PROCENDS
830 XF%=(XL%+XR%)/2:YF%=YF%+4
840 UNTIL XL%=XR%
850 XF%=XP%:YF%=YP%-4
860 REPEAT
870 PLOT77,XF%,YF%:PROCENDS
880 XF%=(XL%+XR%)/2:YF%=YF%-4
890 UNTIL XL%=XR%
900 ENDPROC
```

This program draws a circle and then calls PROCSFILL at the centre of it. As you will see if you RUN it, this fills the circle quite neatly. How PROCSFILL works is easy to see. It starts by setting XF% and YF% to the coordinates of the point at which it was called (XP%, YP%). There is then a REPEAT loop which does a PLOT 77 at XF%, YF% and finds the ends of the line drawn. A new value of XF% is found by averaging the ends of this last line. In addition YF% is incremented by 4 so that the next line drawn will be one line of points further up the screen. This process is repeated until the left and right hand ends of the line coincide. There is then another REPEAT loop which does the same but going down the screen (Lines 86Ø to 89Ø).

You may think that this program is all that is needed to fill any shape. This is not correct.

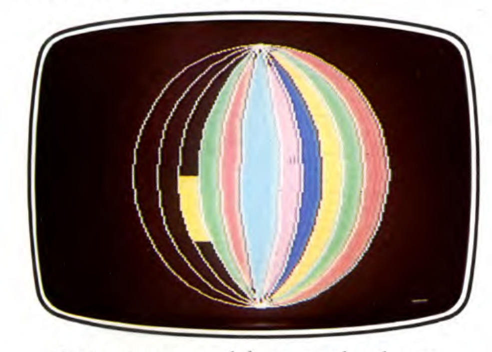
Type in the following line and RUN the program again.

```
45 MOVE-300,-300:DRAW50,-50:
   DRAW300,-300
```

This new line draws a V shape in the circle drawn by the original program, and tries to fill the resulting area using PROCSFILL. Unfortunately, it fails. The reason why is that PROCSFILL cannot detect the fact that the area it is filling splits into two and that it should fill both parts.

Despite these limitations, it is possible to produce some good results with PROCSFILL. Delete Line 60 and Line 45 then type in the following lines and RUN the program:

```
10 MODE2
40 PROCBALL
50 PROCBALLFILL
940 DEFPROCBALL
950 GCOL0,7
960 R%=490
970 PROCCIRCLE(0,0,R%,0,2*PI)
```



but can cope with several colours.

```
980 FORDR%=80 TO 420 STEP80
990 RX%=(DR%*DR%+
    R%*R%)/2/DR%
1000 OL=ASN(R%/RX%)
1010 PROCCIRCLE(DR%-RX%,0,RX%,
     -OL,OL)
1020 PROCCIRCLE(RX%-DR%,0,RX%,
     PI-OL,PI+OL)
1030 NEXT
1040 ENDPROC
1080 DEFPROCBALLFILL
1090 FOR I%=-5 TO 5
1100 GCOL0, (5+I%)MOD6+1
1110 PROCSFILL(I%*80+
     SGNI%*40,0)
1120 NEXT
1130 ENDPROC
```

The additions to the program are two new procedures. The first, PROCBALL, (Lines 940 to 1040) draws a ball on the screen. It does this by calling PROCCIRCLE with the appropriate centres, radii and arc lengths. The ball is filled in by the second new procedure PROCBALLFILL (Lines 1080 to 1130). This calls PROCSFILL at a point in each segment of the ball (Line 1110) and also selects a different colour for each segment at Line 1100.

Now add the following line:

```
60 PROCROTATE
```

and the new PROCedure PROCROTATE:

```
1170 DEFPROCROTATE
1180 K%=0:REPEAT
1190 FORI%=1TO6
1200 VDU19,I%,(I%+K%)MOD6+1,
     0,0,0:NEXT
1210 K%=K%+1
1220 UNTIL INKEY(25)=9
1230 ENDPROC
```

RUN the new program. If it is correct, the ball should appear to rotate. Pressing the TAB key will again end the program. The idea of PROCROTATE is to change all of the red sectors of the ball to green, the green to yellow and so on, and to keep repeating the process, thus producing an apparent rotation. PROCROTATE does the colour-switching using VDU19. It has two loops, a FOR ... NEXT loop (Lines 1190 to 1200) which switches all the colours in the ball, and a REPEAT loop which keeps doing the switching. Again, INKEY is used both to slow down the loop and to detect when the TAB key is pressed. The variable K% in the REPEAT loop is used to keep track of the offset of the physical colour numbers from their actual numbers. The MOD operator in Line 1200 ensures that physical colours in the range 1 to 6 are used (red to cyan).



Complex shapes...

ALL-PURPOSE FILL

Now a procedure which will fill in any shape must be developed. SAVE the program in your computer. Type NEW followed by RETURN and enter the next program:

```
10 MODE2
20 VDU23;8202;0;0;0;
30 PROCSETUP
40 MOVE100,100
50 FORI%=1TO6:DRAWRND(1280),
   RND(1024)
60 NEXT:DRAW100,100
70 PROCFILL(400,400,RND(5),0)
80 IFINKEY(100)< >9:CLS:GCOL0,7:
   GOTO40
90 END
100 DEFPROCSETUP
110 DIML%(100),R%(100),Y%(100),S%(100)
120 OSWORD=&FFF1
130 DIM P% 8
140 X%=P% MOD256
150 Y%=P% DIV256
160 A%=&D
170 ENDPROC
200 DEFPROCENDS
210 CALLOSWORD
220 L%=P%?0+256*P%?1
230 IFL%>=&8000 L%=L%-&10000
240 R%=P%?4+256*P%?5
250 IFR%>=&8000 R%=R%-&10000
260 ENDPROC
290 DEFPROCFILL(XF%,YF%,C%,B%)
300 GCOL0,C%:GCOL0,128+B%
310 N%=-1
320 IFPOINT(XF%,YF%)< >B%ENDPROC
330 PLOT77,XF%,YF%:PROCENDS
340 IFL%=R% PLOT69,XF%,YF%
350 PROCE(L%,R%,YF%-4,-4)
360 PROCE(L%,R%,YF%+4,+4)
370 REPEAT
380 PROCF(L%(N%),R%(N%),Y%(N%),
    S%(N%))
390 UNTIL N%<0
400 ENDPROC
420 DEFPROCF(XL%,XR%,YP%,S%)
430 N%=N%-1
440 IFPOINT(XL%,YP%)< >B%THEN520
450 PLOT77,XL%,YP%
460 PROCENDS
470 IFL%=R% PLOT69,L%,YP%
480 PROCE(L%,R%,YP%+S%,S%)
490 IFL%<XL% PROCE(L%,XL%-8,
    YP%-S%,-S%)
500 IFXR%<R% PROCE(XR%+8,R%,
    YP%-S%,-S%)
510 XL%=R%:IFXL%>=XR% ENDPROC
520 PLOT92,XL%,YP%
530 PROCENDS
540 XL%=R%+8:IFXL%>XR% ENDPROC
```



need the all-purpose fill routine

```
550 IFPOINT(XL%,YP%)<>0 ENDPROC
560 GOTO 450
590 DEFPROCE(LP%,RP%,YP%,SP%)
600 N%=N%+1
610 L%(N%)=LP%:R%(N%)=RP%
620 Y%(N%)=YP%:S%(N%)=SP%
630 ENDPROC
```

To show the impressive results that PROCFILL can produce, there is a simple demonstration program in Lines 10 to 90 of this program. Line 20 gets rid of the flashing text cursor. Lines 40 to 60 draw a random shape, and then Line 70 calls PROCFILL at a point on the screen and with a random foreground colour. If this point is inside the shape, then the inside will fill, otherwise the outside is filled. Line 80 stops the program if TAB is pressed, otherwise it clears the screen and loops back to Line 40.

Note that in Lines 490, 500 and 540, the number 8 appears. This is the number of horizontal graphics units for each point in MODE 2. If you use the program in other MODEs, you must change this number. For MODEs 2 and 5 use 8; for MODEs 1 and 4 use 4 and for MODE 0 use 2.

This is a complicated program. It is, however, split into a number of PROCedures. The way to understand the whole program is to understand what each PROCedure does on its own. PROCSETUP (Lines 100 to 170) is just the usual way of setting up the OSWORD call. But this time Line 110 has been added which DIMensions four arrays L%(), R%(), Y%() and S%() with 101 elements each. These arrays form a stack or list, in which horizontal lines are stored. L%() and R%() hold the x coordinates of the left and right hand ends of the line, Y%() holds its y coordinate and S%() contains the number of graphics units the line is above or below the previous one drawn. The more complicated the shape to be filled, the more elements should be reserved for these arrays. Over a hundred will be enough for all but the most complicated shapes. However, if you are short of memory, you can reduce this number. If enough elements have not been reserved the program will end with the error message 'Subscript at line 610'.

The PROCedure PROCENDS (Lines 200 to 260) is a simplified version of the PROCedure of the same name used earlier. Only the x coordinates of the line are calculated, and this time they are put into the variables L% and R%.

The next PROCedure to understand is PROCE (Lines 590 to 630). The purpose of this is to add a new line to the list of lines kept in the arrays. It is called with values for the parameters LP%, RP%. YP% and SP%. The first thing it does is to increment the variable N% by one. This variable is used to point to the top elements in the list of lines. Next LP%, RP%, YP% and SP% are put into the array elements L%(N%), R%(N%), Y%(N%) and S%(N%). In this way, a new line is added to the list.

The PROCedure that is actually called to fill an area is PROCFILL (Lines 290 to 400). Its parameters are XF% and YF%, the coordinates of the point at which filling is to commence, and C% and B%, the foreground and background colours. In Line 300, these colours are actually set up with GCOL (note that 128 is added to B% only at this point, don't call the PROCedure with 128 already added to B%). Lines 310 to 360 are concerned with setting things up so that the filling process can start. Line 310 sets the pointer N% to −1. Line 320 uses the POINT function to check if the point at which the fill procedure has been called is actually in the background colour. If this is not the case then the PROCedure ends.

Line 330 does a PLOT 77 at XF%, YF% and finds the ends of the line drawn. If both ends coincide then a point must be filled in using the ordinary PLOT 69 statement (Line 340). Now, in Lines 350 and 360, two lines parallel to that drawn are entered into the line list using PROCE. Notice that one is below the line drawn and one above, and that the SP% argument of PROCE is set to minus or plus 4 respectively. The filling now starts properly. Lines 370 and 380 are a REPEAT loop which keeps calling PROCedure PROCF on the top element of the line list UNTIL there are no more elements left in it (which occurs when the entire area is filled).

EXPLAINING PROCF

All that is left to be explained is PROCF (Lines 420 to 560). The arguments of this are XL%, XR%, YP% and S%. This procedure is always called with these parameters set to the top elements of the line list arrays L%(), R%(), Y%() and S%(). These define a line parallel to one that has been filled in. This line on which PROCF is called is conveniently referred to as the 'original line'. The first thing the PROCedure does is to decrement N% thus dropping the top element of the line list. At Line 440, control can branch two ways. If the colour at the left-hand end of the original line (XL%, YP%) is not background, then it must go to Line 520, otherwise it continues at Line 450. First consider the case when the left-hand end of the original line is in the background colour. In this situation, Lines 450 and 460 do a PLOT 77 at this point and find the ends of any line drawn. Line 470 again corrects for any zero length lines by



plotting a single point with PLOT 69, and Line 480 puts a line into the line list parallel to the one drawn. This line is given the same direction (up or down the screen) as the original line by using S% as the last argument of PROCE. However, if the line drawn is longer than the original line then extra lines with length equal to the difference of the two are put into the list with the opposite direction to the original (Lines 490 and 500). Line 510 sets XL% to R% (so the right-hand end of the drawn line becomes the left-hand end of the original). If XL% is now greater than or equal to XR%, the procedure ends, otherwise it continues at Line 520. This is the line to which control passes at the start of the procedure if the point at the left-hand end of the original line is not in the background colour. Lines 520 and 530 use a PLOT 92 fill statement to search to the right for a point in the background colour. If one is found with an x coordinate within the original line, then the program sets XL% to the x coordinate of the point. Before Line 560 loops back to Line 450, a check is made using the POINT function to see if the colour at this point is greater than



zero. This is because points off the screen have colour −1. By doing this check, filling stops at the edges of the screen.

Consider again what PROCF does. It is given a line to fill (the original line), it does this and produces a parallel line for the line list with the same direction as the original line. If the line it draws is longer than the original, then lines are added to the list parallel to the differences, and going in the opposite direction to the original. Last, if any parts of the original line are not in the background colour, they must be missed out, and a search done for any lines inside the original which are in the background cover. These are treated in the same way as the original.

This, then, is a complete general purpose utility for filling in shapes; you can use it in your own programs. Another demonstration of its abilities is given below. Delete Lines 6Ø to 8Ø of the last program and add:

```
3Ø VDU29,64Ø;512;
4Ø PROCSETUP
5Ø PROCSPIRALS
9Ø END
66Ø DEFPROCSPIRALS
67Ø DS=PI/16:RC%=6Ø:D=PI/3
68Ø MOVERC%,Ø
69Ø FORI%=ØTO5:GCOLØ,I%+1
7ØØ DO=I%*D
71Ø PROCSPIRAL(DO)
72Ø MOVERC%*COSDO,RC%*SINDO
73Ø DRAWRC%*COS(DO+D),RC%*SIN
   (DO+D)
74Ø NEXT
75Ø FORI%=2TO8
76Ø O=I%*D−D/2
77Ø XS%=9Ø*COSO:YS%=9Ø*SINO
78Ø PROCFILL(XS%,YS%,I%−1,Ø)
79Ø NEXT
8ØØ ENDPROC
83Ø DEFPROCSPIRAL(DO)
84Ø O=Ø:REPEAT
85Ø R%=RC%*EXP(.4*O)
86Ø XS%=R%*COS(O+DO):YS%=
    R%*SIN(O+DO)
87Ø DRAW XS%,YS%
88Ø O=O+DS
89Ø UNTIL POINT(XS%,YS%)=−1
9ØØ ENDPROC
```

RUN the program. You should see a very pretty pattern. Lines 83Ø to 9ØØ are a PROCedure PROCSPIRAL, which not surprisingly draws a spiral. Line 66Ø is the start of another procedure, PROCSPIRALS. This draws a number of spirals on the screen using PROCSPIRAL, and then uses the PROCedure PROCFILL to fill in each piece of the resulting pattern (Lines 75Ø to 79Ø).

Finally, insert the following lines:

```
6Ø PROCROTATE
93Ø DEFPROCROTATE
94Ø K%=Ø:REPEAT
95Ø FOR I%=1TO6
96Ø VDU19,I%,(I%+K%)MOD6+1,
   Ø,Ø,Ø:NEXT
97Ø K%=K%+1
98Ø UNTIL INKEY(25)=9
99Ø ENDPROC
```

These add the PROCedure DEFPROC ROTATE which was defined in Lines 117Ø to 123Ø of the rotating ball program, and use it to make the spiral appear to spin. If this spinning effect hypnotises you then try learning all the PLOT numbers!

SENDING SECRET MESSAGES

Linking the world of the ancient Greeks with the world of micro electronics and beyond, codes and ciphers can be handled expertly by computer programs

Most people use codes in their everyday life. Anyone asking for a number 8 screw or a size 5 football is employing a code. Similarly, a nuclear scientist's seemingly incomprehensible equations are only using a shorthand code to represent complex relationships. This could be done equally well in plain words but it would often take a good deal longer.

In all the above examples, the emphasis is on the improvement of data communication rather than the concealment of information, and another related use of codes is to save money or storage space. For example, a company may issue contracts which use standard clauses or phrases. If details of these agreements are to be stored on tape or disk, a lot of valuable space can be saved by number coding the most often repeated sections. This has become known as data compression.

Similarly, the standard responses used in such adventure games as *The Hobbit* or *Valhalla* are usually coded in order to economize on storage. *INPUT*'s own text compressor (pages 628 to 636 and following articles) is an example of just such a coding system.

The Greeks invented the science of sending secret messages, so it is not surprising that the formal name for coding—cryptography—should come from two Greek words: *kryptos* (secret) and *graphos* (writing). The terms *code* and *cipher* actually have slightly different meanings related to the two ways in which messages may be sent. When information is translated letter by letter, this is called enciphering. On the other hand, if whole words or groups of words are transformed into other words or numbers by referring to a special dictionary, this is called coding. In practice, the term code is used generally to refer to both codes and ciphers.

SECRET CODES

The coding and decoding of secret messages was once an area confined mainly to the military, or at least the Intelligence Services. Today, however, the extended use of the public telephone lines and satellite channels for the transmission of commercially sensitive data has increased the need for encoding.

At the dawn of the computer revolution, IBM comptometers were used to break Second World War codes. Since then, each advance in computer technology has been eagerly monitored by spymasters and code breakers. Today, with good programming and the right cipher, a home micro can match any conventional cryptography machine. This is the first of two articles which show how to use your computer to produce secret coded messages, using several different methods which, like spying itself, are of different levels of sophistication.

Even if you are not an international agent, the methods used are interesting in themselves, and you can always use them to send coded messages to other friends with computers. In fact, there is a message hidden somewhere in this article.

DISTANCE CODES

The seemingly haphazard pattern of symbols shown **right** is, in fact, an example of the distance code. As its name suggests, this is a code which is based on the distance of a particular symbol from a given point.

This type of code was used more than 2,000 years ago by the Greek General Lysander. The distances of the notches from the belt buckle of one of the slaves spelled out a secret message which helped the General to defeat the Persian Empire.

You can use a form of distance code simply by ranging the letters of the alphabet across the top line of a lined pad and making out the message as shown in **Figures 1** and **2**. While you can see the letter key on the top line, the message is easy to understand. Once this is removed, however, decoding the information is not so easy.

To make deciphering even more tricky, rotate the letter key at the top of the page (see **Figure 3**). You can, for instance, start with N and when you reach Z simply begin once more with A. This is called cyclic rotation.

The first program uses this method to produce a coded version of your plain text, providing you don't leave spaces between the words. If you wish actually to send the message you'll need access to a printer, and to make sure that the recipient of the text knows which order to use when decoding.

S

```
20 BORDER Ø: PAPER Ø: INK 7: CLS
30 PRINT TAB 8; "□Distance code□":
   PRINT
40 PRINT INK 2; PAPER 7; FLASH 1;AT 6, 1Ø;
   "□□WARNING□□": PRINT
50 PRINT "Don't leave spaces between words"
60 PRINT : PRINT
70 INPUT "What is your message ?"'a$
80 FOR i=1 TO 4ØØ: NEXT i: CLS
90 FOR i=1 TO LEN (a$)
100 LET b$=a$(i)
110 LET v=CODE (b$)-96
120 IF v<=32 THEN PRINT TAB v; INK
   6;"*": GOTO 15Ø
```



- CODES AND CIPHERS
- CRYTOGRAPHY AND THE ANCIENT GREEKS
- SECRET CODES AND THEIR APPLICATIONS
- PRODUCING YOUR OWN SECRET CODES
- DISTANCE CODES
- THE ST. CYR CIPHER
- MORSE CODE

```
130 LET v=v-26
140 PRINT TAB (v); INK 6;"*"
150 NEXT i
```




```
30 PRINT "♡▣>□▣DISTANCE CODE"
50 PRINT "▣▣DON'T LEAVE
   SPACES":
   PRINT "BETWEEN WORDS !"
70 PRINT "▣WHAT IS YOUR
   MESSAGE":INPUT A$
80 PRINT "♡"
90 FORI=1 TO LEN(A$)
100 B$=MID$(A$,I,1)
110 V=ASC(B$)-45
120 IF V<=32 THEN PRINT
   TAB(V);"*":GOTO 150
130 V=V-26
140 PRINT TAB(V);"*"
150 NEXT I
```



```
20 MODE1:VDU 19,0,3,0,0,0,19,7,
   4,0,0,0
30 PRINTTAB(13);
   "□DISTANCE CODE"
40 PRINTTAB(17);
   "WARNING": PRINT"
50 PRINTTAB(2);
   "DO NOT□LEAVE SPACES
   BETWEEN WORDS"
60 PRINT"
```





Horizontal distance code



Vertical distance code

```
70 INPUT"WHAT IS YOUR
   MESSAGE",A$
80 TIME=0:REPEAT UNTIL TIME
   >300: CLS
90 FOR I=1 TO LEN(A$)
100 B$=MID$(A$,I,1)
110 V=ASC(B$)-45
120 IF V<=32 THEN PRINT TAB(V);
   "*":GOTO 150
130 V=V-26
140 PRINTTAB(V)"*"
150 NEXT I
```



```
20 CLS
30 PRINT@9,"DISTANCE CODE"
40 PRINT@140,"WARNING":PRINT
50 PRINT "DON'T LEAVE SPACES BETWEEN
   WORDS"
60 PRINT:PRINT
70 PRINT"WHAT IS YOUR MESSAGE":
   INPUTA$
80 FOR I=1 TO 600:NEXT:CLS
```

```
90 FOR I=1 TO LEN(A$)
100 B$=MID$(A$,I,1)
110 V=ASC(B$)-45
120 IF V<=32 THEN PRINTTAB(V)
   "*":GOTO150
130 V=V-26
140 PRINTTAB(V)"*"
150 NEXT
```

The program works by setting up a loop and using the MID$ facility to evaluate the ASCII values of each letter of plain text (Lines 90, 140). A previous article (page 420) showed how the ASCII function allows letters to be represented by numbers. For most home computers, except the ZX81, the letters take similar values. The use of string functions like MID$ has also been discussed in a previous article (pages 202–207).

Once the message has been converted into a series of numbers, it is easy to encode by using a straightforward linear transformation. In the case of the Commodore 64, for example, the plain text letter V is translated to the ASCII equivalent of V, less 26 (Line 130).

It only remains to use the TAB function to print out the asterisk at the required distance from the right-hand side of the screen (Line 120) and the process of coding is complete.

USING THE CODE

Although the distance code may seem a little too simple to be effective, it does have a number of factors in its favour. In the first place, before you can successfully decipher a code, you must first recognize that the code actually exists. And because it is so easy to disguise a pattern of apparently random dots (or asterisks) in an otherwise harmless sketch there's a good chance that a coded message of this type will pass unnoticed.

During the last war, enemy agents used this trick. On closer inspection an innocent looking picture of a garden revealed that pegs on a clothes line spelled out a secret message.

One way to make the distance code more difficult to crack is to restructure the program so that the letter key is truly randomized. As it stands, an expert who realises that he is dealing with a distance cipher needs to try, at most, 26 combinations before solving the problem. If the order of letters in the key is random, however, the number of possible combinations increases enormously.

THE ST. CYR CIPHER

It was the Romans who took over from the Greeks as master cryptographers. Julius Caesar invented a straightforward substitution cipher in which each letter was replaced by the letter three places forward in the alphabet. Here A becomes D, B becomes E etc. At the end of the alphabet X becomes A, Y becomes B and Z takes the character C. Using this method, the message THE RUSSIANS ARE COMING enciphers to:

WKH UXVVLDQV DUH FRPLQJ

As you will see later, Caesar's code is a special case of something called the St. Cyr Cipher, and you can easily check this result by running the next program and using 3333333 as the number key.

With the demise of the Roman Empire, developments in cryptography ceased, and despite the increased use of codes in the sixteenth and seventeenth centuries, it was not until the nineteenth century that the French military academy at St. Cyr produced a significant improvement to Caesar's code. The St. Cyr cipher is brilliantly simple. It is made up of three alphabets on a sliding scale. The bottom alphabet is plain text and the cipher equivalents are taken from the top line. From the starting point shown in the diagram, INPUT would be encrypted as AFHML. One of the advantages of the St. Cyr cipher is that a different alphabet equivalent can be used to code each letter. This can make code cracking very difficult.

In the St. Cyr cipher program, this facility has been incorporated into a number key in order to provide extra security. Anyone with access to a program listing will still not be able to solve the code, unless, of course, they also know the seven-figure secret numbers.



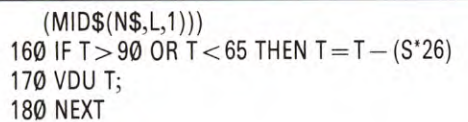
Julius Caesar's special code

```
20 BORDER 0: PAPER 0: INK 7: CLS
25 POKE 23658,8
30 PRINT TAB 10;"ST-CYR CIPHER": PRINT
40 PRINT INK 2; PAPER 7; FLASH 1;AT 6,10;
   "□□WARNING□□"
50 PRINT "Don't leave spaces between words"
60 PRINT : PRINT
70 PRINT "ENTER 1 if you wish to encode"
80 PRINT "ENTER -1 if you wish to decode"
90 INPUT s
100 INPUT "enter your
   message"'a$
110 PAUSE 50: CLS
120 INPUT "enter seven figure
   number key"'n$
130 PAUSE 50: CLS
140 FOR k=1 TO LEN a$
150 LET l=k-INT (k/7)*7+1
160 LET t=CODE (a$(k))+
   (s*VAL (n$(l)))
170 IF t>90 OR t<65 THEN LET
   t=t-(s*26)
180 PRINT CHR$ (t);
190 NEXT k
```

```
30 PRINT "■■>□■ST-CYR CIPHER"
70 PRINT "■■(+1) ENCODE
   MESSAGE"
80 PRINT "(-1) DECODE MESSAGE"
90 INPUT "■■ENTER 1
   OR-1■";S
100 PRINT "■TYPE IN YOUR
   MESSAGE":INPUT A$
120 PRINT "■ENTER SEVEN
   FIGURE":INPUT "NUMBER KEY■";N$
130 PRINT "■"
140 FOR K=1 TO LEN(A$)
145 M$=MID$(A$,K,1):IF M$<"A" OR
   M$>"Z" THEN PRINT M$;:
   GOTO 190
150 L=K-INT(K/7)*7+1
160 T=ASC(MID$(A$,K,1))+(S*VAL
   (MID$(N$,L,1)))
170 IF T<65 OR T> 90 THEN
   T=T-(S*26)
180 PRINT CHR$(T);
190 NEXT K:PRINT
```



```
20 MODE1:VDU 19,0,3,0,0,0,19,7,
   4,0,0,0
30 PRINTTAB(13)"ST-CYR CIPHER":
   PRINT
40 PRINT'TAB(16);"WARNING"
50 PRINTTAB(2);"DO NOT LEAVE SPACES
   BETWEEN WORDS"''
60 PRINT''
70 PRINT"ENTER 1 IF YOU WISH TO ENCODE
   MESSAGE"
80 PRINT"ENTER -1 IF YOU WISH TO
   DECODE MESSAGE"
90 INPUTS:PRINT
100 INPUT"TYPE IN YOUR MESSAGE",A$
110 TIME=0:REPEAT UNTIL TIME>150:
   CLS
120 INPUT"ENTER SEVEN FIGURE NUMBER
   KEY",N$
130 FOR K=1 TO LEN(A$)
140 LET L=K-INT(K/7)*7+1
150 T=ASC(MID$(A$,K,1))+(S*VAL
```



```
   (MID$(N$,L,1)))
160 IF T>90 OR T<65 THEN T=T-(S*26)
170 VDU T;
180 NEXT
```

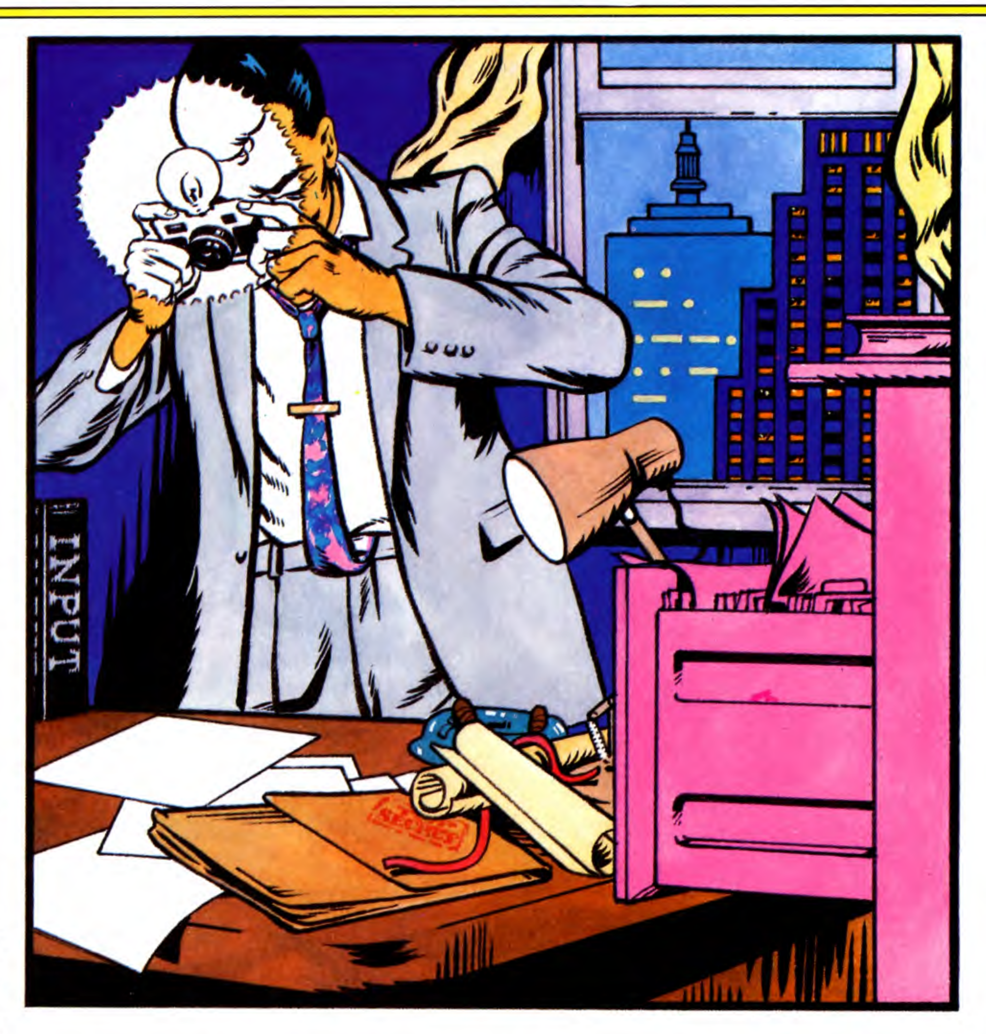
```
20 CLS
30 PRINT@9,"ST-CYR CIPHER":PRINT
40 PRINT@140,"WARNING":PRINT
50 PRINT"DON'T LEAVE SPACES BETWEEN
   WORDS"
60 PRINT:PRINT
70 PRINT"ENTER □□1 IF YOU WISH TO
   ENCODE"
80 PRINT"ENTER -1 IF YOU WISH TO
   DECODE"
90 INPUT S
100 INPUT"ENTER YOUR MESSAGE";A$
110 FORK=1TO1500:NEXT:CLS
120 INPUT"ENTER SEVEN FIGURE NUMBER
   KEY□□□";N$
130 FORK=1TO1500:NEXT:CLS
140 FOR K=1 TO LEN(A$)
150 L=K-INT(K/7)*7+1
160 T=ASC(MID$(A$,K,1))+(S*VAL
   (MID$(N$,L,1)))
170 IF T>90 OR T<65 THEN T=T-(S*26)
180 PRINTCHR$(T);
190 NEXT
```

The operation of this program is very similar to the distance code program. As each letter of plain text is read, it is first converted into its ASCII code numbers equivalent. The VAL function is then employed to increment this figure by an amount dictated by the relevant digit in your number key (Line 160). After a check to ensure that the result lies in an acceptable range (Line 170), the CHR$ facility is used to output the coded message.

During the early days of June 1944, the following message might have needed coding: TROOPSCAPTUREPEGASUS BRIDGE. Using the program with a number key 382110̸5, the encoded text becomes: BTPPPXFIRUVRJSMIBTUXEZKEHE. By using an indicator variable S which can take values +1 or −1, it is possible to use the same program for decoding.

The seven-figure number key is a matter of personal choice. However, as these figures often have to be committed to memory, it's a good idea to use something familiar, such as a telephone number. As there are ten million possible different values for the number key, the St. Cyr cipher is already hard to decode. However, extra difficulty can be introduced by coding twice—putting the already coded message through the program for a second time.

Consider the message TOM IS A SPY.



Using number keys 382110̸5 and 2568723 the enciphering/deciphering sequence becomes:

Text	Number Key	
TOM IS A SPY		↑
	382110̸5	Code
BQNJSFVXA		
	2568723	↓
GWVQIXCG		↑
	2568723	
BQNSFVXA		Decode
	382110̸5	
TOMISASPY		↓

MORSE CODE

Even the most secure code is of little use if it cannot be transmitted quickly. Napoleon made some attempts to overcome the problem of speed by building signal towers throughout France. A primitive version of semaphore was used to pass messages from tower to tower.

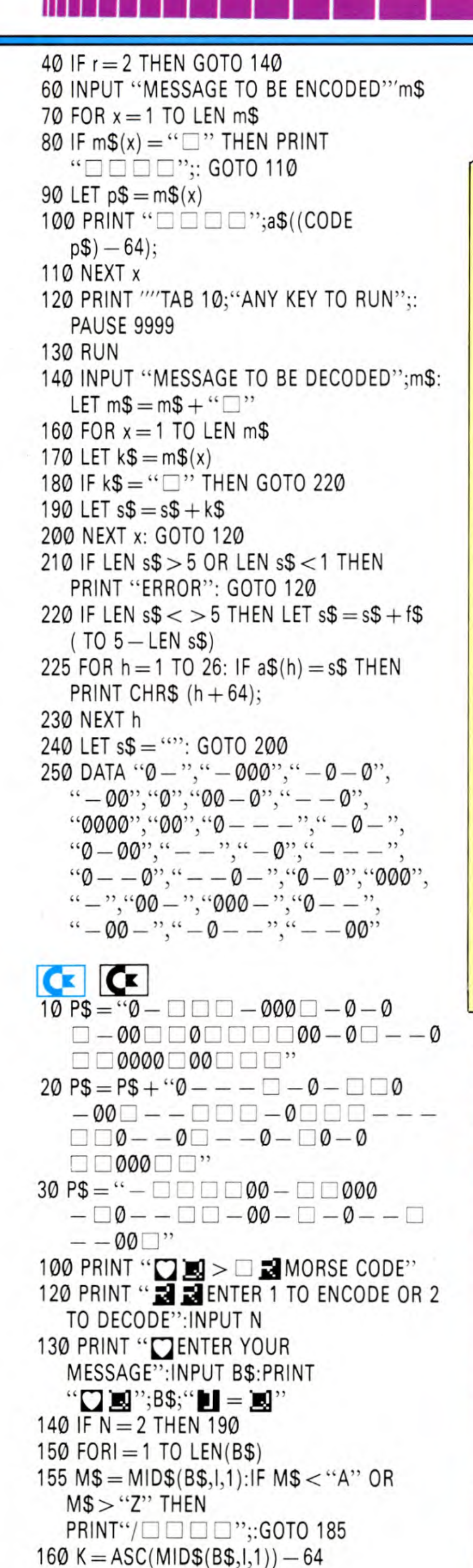
However, it was the invention of the electromagnetic telegraph and accompanying substitution cipher known as Morse Code in the 1830's that really speeded up the sending of secret messages. The American inventor Morse devised a code in which dots and dashes replace letters, as shown in the table:

The Morse Code program will either encipher plain text, or decode a series of dots and dashes into the appropriate letters. As an example, the well-known distress signal SOS becomes /··· /--- /···/ and the slightly more elaborate text ESCAPE AT MIDNIGHT is: /· /··· /-·-· /·- /·--· /· /·- /-/-- /·· /-·· /-· /·· /--· /···· /-

Try the two passages of text in the next program. Working in the reverse direction, if you run the program and enter -- 0̸- -0̸-- - 0̸0̸0̸--0̸--***** the word MAYDAY will appear on the TV screen. Asterisks are used to tell the computer that the message is complete except for the Acorn and Dragon.



```
10 BORDER 0: PAPER 0: INK 7: CLS : DIM
   a$(26,4): LET S$="": LET
   f$="□□□□□□"
15 POKE 23658,8
20 FOR x=1 TO 26: READ a$(x): NEXT x
30 INPUT "Encode (1) OR Decode (2) ";r
```



```
40 IF r=2 THEN GOTO 140
60 INPUT "MESSAGE TO BE ENCODED"'m$
70 FOR x=1 TO LEN m$
80 IF m$(x)="□" THEN PRINT
   "□□□□";: GOTO 110
90 LET p$=m$(x)
100 PRINT "□□□□";a$((CODE
   p$)-64);
110 NEXT x
120 PRINT ''''TAB 10;"ANY KEY TO RUN";:
   PAUSE 9999
130 RUN
140 INPUT "MESSAGE TO BE DECODED";m$:
   LET m$=m$+"□"
160 FOR x=1 TO LEN m$
170 LET k$=m$(x)
180 IF k$="□" THEN GOTO 220
190 LET s$=s$+k$
200 NEXT x: GOTO 120
210 IF LEN s$>5 OR LEN s$<1 THEN
   PRINT "ERROR": GOTO 120
220 IF LEN s$<>5 THEN LET s$=s$+f$
   ( TO 5-LEN s$)
225 FOR h=1 TO 26: IF a$(h)=s$ THEN
   PRINT CHR$ (h+64);
230 NEXT h
240 LET s$="": GOTO 200
250 DATA "0-","-000","-0-0",
   "-00","0","00-0","--0",
   "0000","00","0---","-0-",
   "0-00","--","-0","---",
   "0--0","--0-","0-0","000",
   "-","00-","000-","0--",
   "-00-","-0--","--00"
```

```
10 P$="0-□□□-000□-0-0
   □-00□□0□□□□00-0□--0
   □□0000□00□□□"
20 P$=P$+"0---□-0-□□0
   -00□--□□□-0□□□---
   □□0--0□--0-□0-0
   □□000□□"
30 P$="-□□□□00-□□000
   -□0--□□-00-□-0--□
   --00□"
100 PRINT "♥■>□■MORSE CODE"
120 PRINT "■■ENTER 1 TO ENCODE OR 2
   TO DECODE":INPUT N
130 PRINT "♥ENTER YOUR
   MESSAGE":INPUT B$:PRINT
   "♥■";B$;"■=■"
140 IF N=2 THEN 190
150 FORI=1 TO LEN(B$)
155 M$=MID$(B$,I,1):IF M$<"A" OR
   M$>"Z" THEN
   PRINT"/□□□□";:GOTO 185
160 K=ASC(MID$(B$,I,1))-64
170 T=1+(K-1)*5
180 PRINT "/" MID$(P$,T,5);
185 NEXT I:GOTO 280
```



What are the practical uses of codes?

Aside from the traditional uses of codes in the shady world of subterfuge, codes are finding their way increasingly into our daily lives.

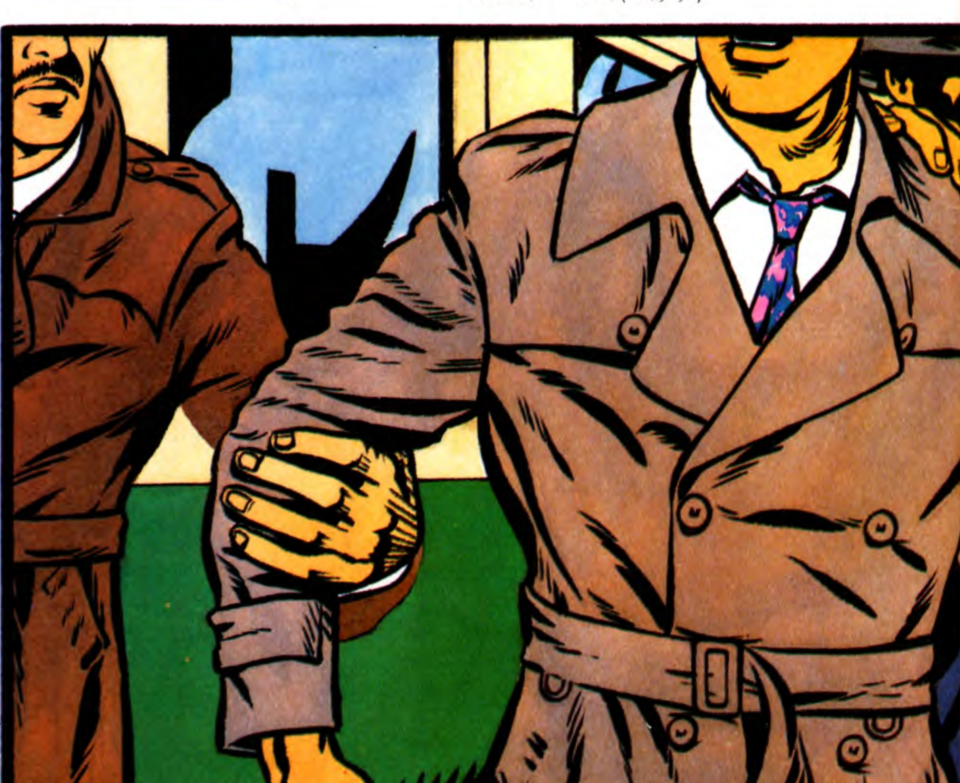
Nearly everyone who has a bank account now has a card which enables money to be withdrawn from an automatic till. The card has its own code. Likewise, each cheque has codes written along its bottom edge, related to the branch, account and cheque number.

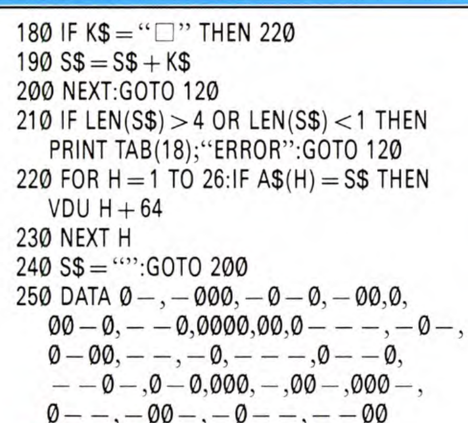
For many years stock control has been made easier by using computers, with each item being assigned a code number. With the introduction of bar codes—the black and white stripes found on shopping items—prices can be automatically read, and the bill compiled at the same time as the stock totals are adjusted.

This is moving towards the 'cashless society' where money is moved around without being touched. This increasing computer control may have its advantages—certainly in terms of avoidance of theft and fraud. Eventually, it may be that every shop has access to your credit rating.

```
190 FOR I=1 TO LEN(B$)
200 IF MID$(B$,I,1)<>"/" THEN NEXT I
210 D$=LEFT$(B$,I-1)+"□□□□
   ":B$=MID$(B$,I+1)
220 IF D$="*" THEN 280
230 FOR V=1 TO 26:W=1+(V-1)*5
240 IF LEFT$(D$,5)<>MID$(P$,W,5)
   THEN 260
250 PRINT CHR$ (64+INT(W/5)+1);:GOTO
   270
260 NEXT V:PRINT "□";
270 IF B$<>"" THEN 190
280 PRINT
```

```
10 MODE6:DIM A$(26)
20 FOR X=1 TO 26:READ A$(X):NEXT X
30 INPUT''"□□□□□□□ENCODE OR
   DECODE(1/2)",R
40 IF R=2 THEN GOTO 140
50 PRINTTAB(7);"MESSAGE TO BE
   ENCODED";SPC(14);
60 INPUT M$
70 FOR X=1 TO LEN(M$)
80 IF MID$(M$,X,1)="□" THEN PRINT
   "□□□□□□";:GOTO 110
90 P$=MID$(M$,X,1)
100 PRINT "□";A$(ASC(P$)-64);
110 NEXT
120 PRINT'''''TAB(13);"ANY KEY TO
   RUN";:A$=GET$
130 RUN
140 PRINTTAB(7);"MESSAGE TO BE
   DECODED";SPC(14);
150 INPUT M$:M$=M$+"□"
160 FOR X=1 TO LEN(M$)
170 K$=MID$(M$,X,1)
```





```
180 IF K$="□" THEN 220
190 S$=S$+K$
200 NEXT:GOTO 120
210 IF LEN(S$)>4 OR LEN(S$)<1 THEN
    PRINT TAB(18);"ERROR":GOTO 120
220 FOR H=1 TO 26:IF A$(H)=S$ THEN
    VDU H+64
230 NEXT H
240 S$="":GOTO 200
250 DATA Ø–,–ØØØ,–Ø–Ø,–ØØ,Ø,
    ØØ–Ø,––Ø,ØØØØ,ØØ,Ø–––,–Ø–,
    Ø–ØØ,––,–Ø,–––,Ø––Ø,
    ––Ø–,Ø–Ø,ØØØ,–,ØØ–,ØØØ–,
    Ø––,–ØØ–,–Ø––,––ØØ
```

```
10 CLS:DIMA$(26)
20 FOR X=1 TO 26:READ A$(X):NEXT X
30 PRINT@1ØØ,;:INPUT"ENCODE OR
    DECODE(1,2) ";R
40 IF R=2 THEN 140
50 PRINT"□□□MESSAGE TO BE
    ENCODED"
60 INPUT M$
70 FOR X=1 TO LEN(M$)
80 IF MID$(M$,X,1)="□" THEN PRINT
    "□□□□□";:GOTO110
90 P$=MID$(M$,X,1)
100 PRINT"□";A$(ASC(P$)–64);
110 NEXT
120 PRINT:PRINT:PRINTTAB(7);"ANY KEY TO
    RUN"
130 IF INKEY$="" THEN 130 ELSE RUN
140 PRINT"□□□MESSAGE TO BE
    DECODED"
150 INPUT M$:M$=M$+"□":PRINT:
    PRINT
```



Morse Code on screen

```
160 FOR X=1 TO LEN(M$)
170 K$=MID$(M$,X,1)
180 IF K$="□" THEN 210
190 S$=S$+K$
200 NEXT:PRINT:GOTO 120
210 IF LEN(S$)>4 OR LEN(S$)<1 THEN
    PRINT"□";:GOTO200
220 FOR H=1 TO 26:IF A$(H)=S$ THEN
    PRINTCHR$(H+64);
230 NEXT
240 S$="":GOTO200
250 DATA Ø–,–ØØØ,–Ø–Ø,–ØØ,Ø,
    ØØ–Ø,––Ø,ØØØØ,ØØ,Ø–––,–Ø–,
    Ø–ØØ,––,–Ø,–––,Ø––Ø,
    ––Ø–,Ø–Ø,ØØØ,–,ØØ–,ØØØ–,
    Ø––,–ØØ–,–Ø––,––ØØ
```

In the first part of this program a 130 character string of dots and dashes is set up to represent the morse equivalent of the alphabet in sequential order (Lines 1Ø–3Ø on the Commodore and 25Ø on Acorn, Dragon and Spectrum). While the minus sign provides a good symbol for a dash, it is best if an asterisk (zero in the Acorn and Commodore) or graphics symbol is used for the dots instead of the full stop.

A field of five characters has been allowed for each letter, whereas four would really be adequate for the full alphabet. This is to give anyone wishing to extend the program the opportunity of introducing numbers—which have longer codes.

The coding section is very similar to the first two programs. Each letter of plain text is read in turn and converted to a number between 1 and 26.

The number equivalent is then scaled by five and the appropriate sub-string of dots and dashes is printed (Line 18Ø on the Commodores and Line 1ØØ on Spectrum, Dragon and Acorn).

For the deciphering routine which makes up the final part of the program, a search technique is used (Lines 23Ø–25Ø on the Commodores. Line 22Ø on Acorn and Dragon. Line 225 on the Spectrum).

After reading the Morse signals, the computer will search the string until it finds an identical sub-string. Once this is achieved, it is a straightforward matter to transform the string location into the corresponding ASCII code number. The CHR$ facility is again used to output the required result.

In the second part of this article about codes, you'll see how to decode the simpler transpositions and ciphers and learn about multiplication codes, and codes that are used commercially.

CLIFFHANGER: TUNING IN

It's time to turn on and tune in before Willie drops out. Yes, *INPUT* has gone back to the 60s—the 1560s—for a tune that will strike a chord with Cliffhanger

No game would be complete without an amusing little tune and Cliffhanger plays Greensleeves at appropriate intervals. Now you may think that Greensleeves is not an entirely suitable choice for a game about hill climbing. But according to fable Greensleeves was written by Henry VIII for his wife Anne Boleyn. She later came to a sticky end as will Willie if he makes a slip. Also it is out of copyright. This is important. You wouldn't want to have to pay a royalty every time you played the game.

The following machine code routine uses the Spectrum's BEEPER routine in ROM:

```
        org 60000
        ld ix,57359
msk     ld b,19
tune    push bc
        ld d,(ix+1)
        ld e,(ix+0)
        ld h,(ix+3)
        ld l,(ix+2)
        push ix
        call 949
        pop ix
        ld de,4
        add ix,de
        pop bc
        djnz tune
        ret
```

This routine will play a tune, any tune. But it does need some data to tell the routine what to play. The following BASIC program carries the data necessary to play Greensleeves, which is POKEd into a data table as before:

```
5 CLEAR 57358
10 FOR n=57359 TO 57434 STEP 2: READ
   a:POKE n+1,INT (a/256): POKE
   n,a-(256*INT (a/256)): NEXT n
20 DATA 98,1460,233,1223,131,1086,220,
   964,78,908,147,964,261,1086,110,1297,
   131,1642,49,1460,110,1297,233,1223,98,
   1460,147,1460,44,1642,98,1460,220,1297,
   92,1548,220,1959
```

PLAYING OUR TUNE

The tune routine starts by loading IX with 57359. The IX register pair is going to be used as a pointer and it is set to the beginning of the tune data table.

Then B is loaded with 19. The B register is used as another pointer and 19 is the number of notes in the tune to be played—there are 19 notes in the phrase of Greensleeves. The value of this counter is stored by PUSHing it onto the stack by PUSH BC. C has to be PUSHed along with B as there are no instructions that put the value of single registers onto the stack. It is a fact that a PUSH always works on register pairs.



But what, you may say, is the point in PUSHing this counter onto the stack when neither the B or the C register is used in the routine before the value is POPped back off the stack again? A good question, but there is a ROM CALL in this routine and it might want to use the B register. And though ROM routines might alter the values in any particular register they always leave the stack in the same condition they found it. So the rule is: if in doubt, PUSH it.

DE is then loaded with the first number from the data table and HL with the second. If you look at the BASIC program you will see that each number occupies two bytes in the table, though some are less than 255.

The numbers are in pairs. The first number of each pair specifies the duration of the note. In fact, it specifies the number of cycles generated by BEEP. So the number put in here is the frequency multiplied by the length of time, in seconds, you want the note to last.

The second number controls the pitch. And you work out what it should be by the following formula: 437,500/f−30.125 where f is the frequency.

The registers have to be loaded one at a time because with indexed addressing there is no instruction which loads the register pairs.

The IX register pair are then PUSHed to preserve the pointer while the ROM routine is CALLed. The CALL 949 is the instruction that calls the BEEPER routine. This uses the parameters in HL and DE and makes the appropriate sound accordingly.

The point is then POPped back off the stack into the IX register and it is advanced to the start of the next note by loading 4 into DE and adding. This has to be done with an ADD IX,DE as there is no instruction which adds a number directly to the IX register. You could use four INC IX instructions, but that would take considerably longer.

The BC value is then POPped back off the stack and restored to the register. The DJNZ instruction then decrements the B counter and, if it hasn't reached zero yet, it jumps back to output the next note.

To test your routine, use RANDOMIZE USR 58277.

TUNING UP

It is unlikely that Henry VIII wrote Greensleeves in the form given here. Some of his appetites were pretty basic, but it is unlikely that he had much of an appetite for BASIC.

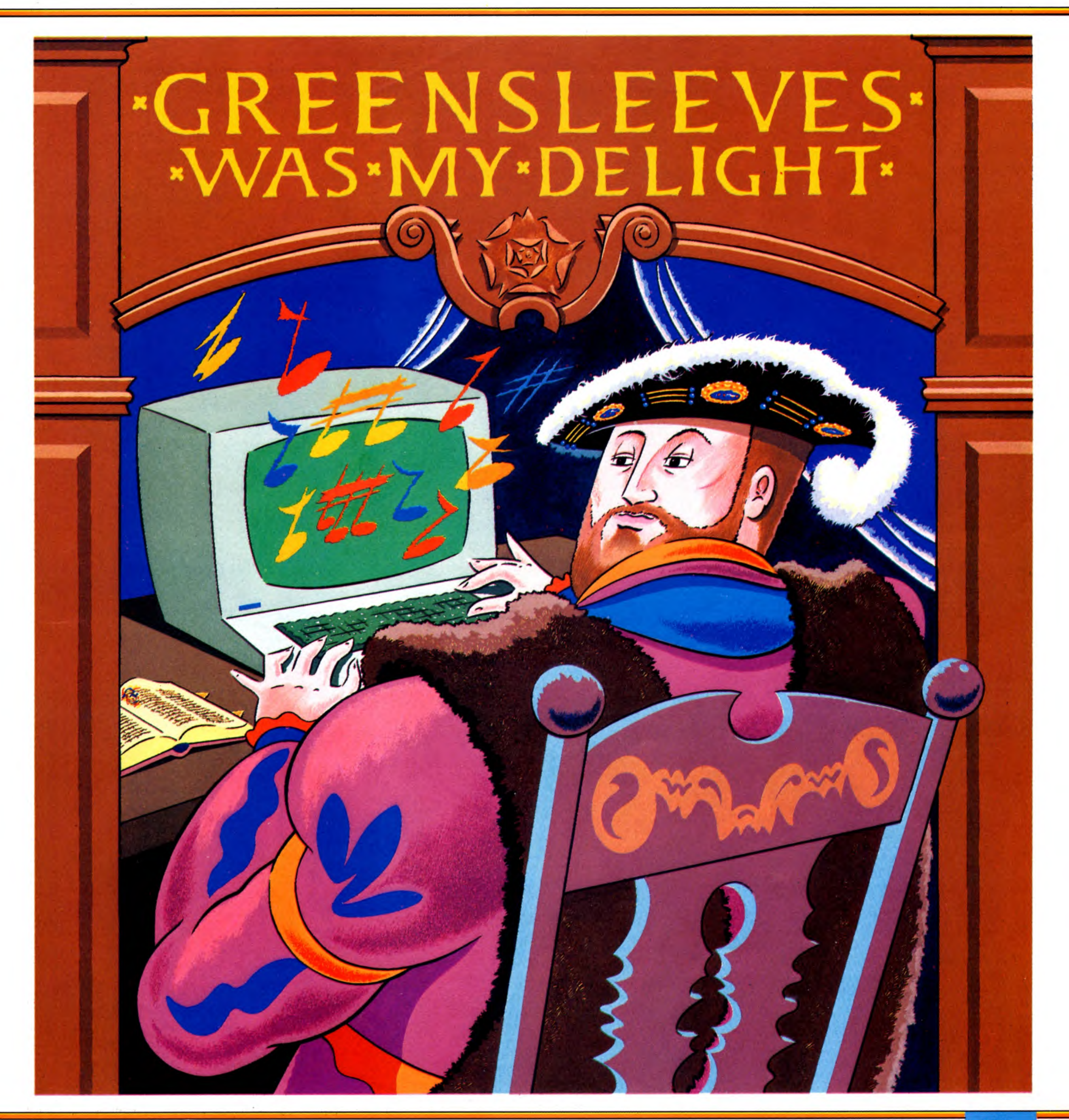
Still any tune can be translated into suitable parameters to be output to the BEEPER routine using the machine code routine here. But, remember, if you are using a different tune you have to change the counter in B.



The routine that plays the tune is set aside from the main routine because it is going to be

The 'CLIFFHANGER' listings published in this magazine and subsequent parts bear absolutely no resemblance to, and are in no way associated with, the computer game called 'CLIFF HANGER' released for the Commodore 64 and published by New Generation Software Limited.

- PICKING A TUNE
- COUNTING THE NOTES
- WORKING OUT THE PITCH
- CHANGING YOUR TUNE
- CLOCKING THE OUTPUT



called several times from several different parts of the program. But as you wait after the title page has been displayed, the following wedge is insert there:

```
ORG  16473
LDA  #$13
JSR  $4Ø96
RTS
```

This loads the accumulator with 13 hex, 19 decimal—there are 19 notes in the initial phrase of Greensleeves. Then it jumps to the tune playing subroutine at 16,534.

```
ORG  16534
CMP  #$13
BCC  $4Ø9D
LDA  #$13
NOP
NOP
NOP
NOP
STA  $FB
LDY  #$18
LDA  #$ØØ
STA  $D4ØØ,Y
DEY
BNE  $4ØA6
NOP
LDA  #$ØF
STA  $D418
NOP
LDA  #$ØØ
STA  $D4Ø5
LDA  #$FØ
STA  $D4Ø6
LDY  #$ØØ
LDA  $436Ø,Y
STA  $D4Ø1
INY
LDA  $436Ø,Y
STA  $D4ØØ
INY
STY  $FC
LDA  #$ØØ
TAX
TAY
JSR  $FFDB
LDY  $FC
NOP
LDA  $436Ø,Y
INY
STY  $FC
STA  $FD
NOP
LDA  #$11
STA  $D4Ø4
NOP
JSR  $FFDE
CMP  $FD
BNE  $4ØE8
NOP
LDY  $FC
LDA  #$ØØ
STA  $D4Ø4
NOP
DEC  $FB
BNE  $4ØBF
STA  $D418
RTS
```

FAIL SAFE

The first little module in this program is a fail-safe device. When this routine is called from some other part of this main program you may forget to load the accumulator appropriately. As there are only enough data to play 19 notes, there is a check to make sure that the routine is not being called on to play any more.

So the contents of the accumulator are compared with 19. The Branch on Carry Clear branches over the next instruction if the carry flag is not set—that is, the contents of the accumulator are less than or equal to 19.

But if the contents of the accumulator are more than 19, the branch does not occur and the accumulator is loaded with 19. This means that the routine can be called on to play less than 19 notes, but not more.

The number of notes to be played is then stored in FB so that the accumulator can be used to initialize the SID chip.

THE SID CHIP

The Commodore's 6581 SID chip controls the synthesis of sound and music. Locations 54,272 to 54,296 are in the SID chip and when you are going to use it, the first thing that has to be done is to clear it out by putting zeros into its 24 locations.

So 24—or 18 hex—is loaded into the Y index register and Ø is loaded into the accumulator. Then zero is stored in 54,272—D4ØØ in hex—offset by the contents of Y, which gives 54,296. Y is then decremented and the processor branches back.

The last SID location—54,296 or D418—controls the overall volume of the sound output. Maximum volume is given by storing 15 in this location.

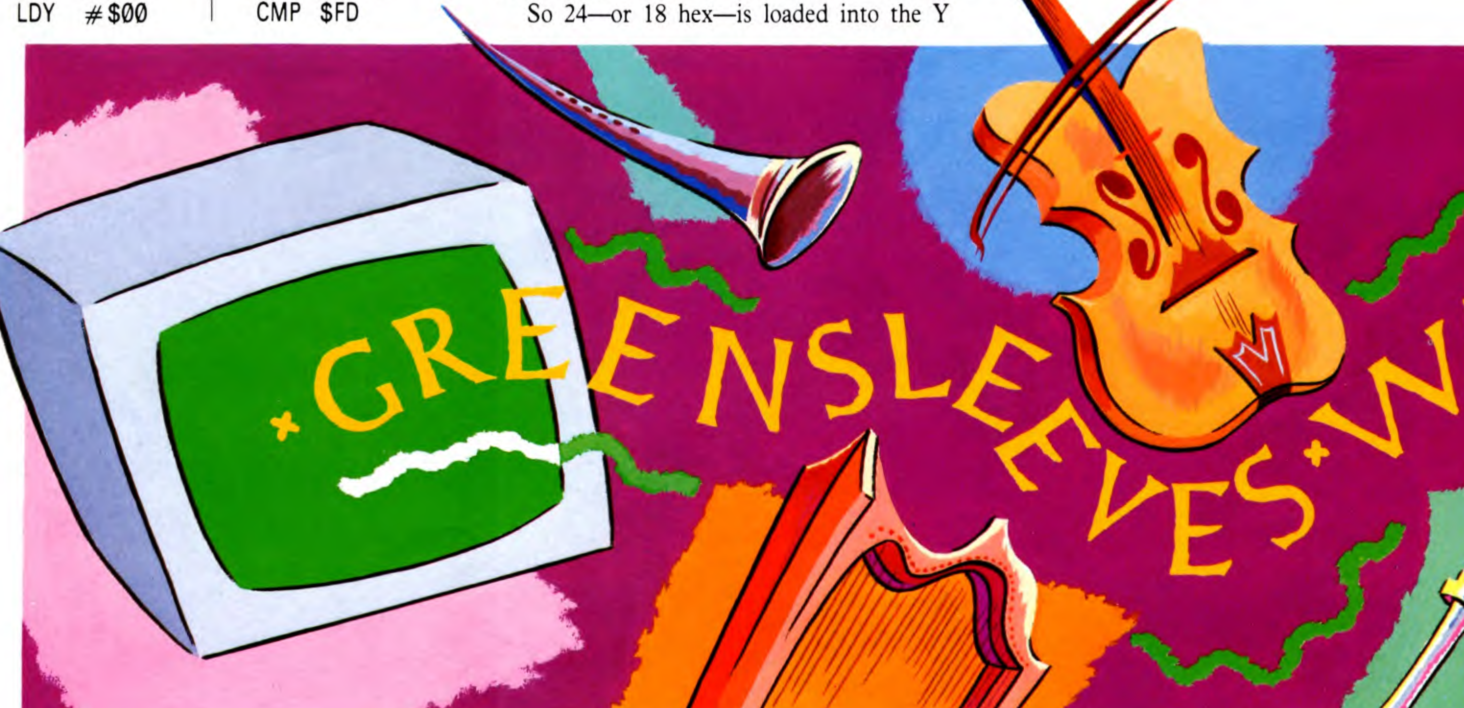
THE ENVELOPE

The way the volume of a sound varies throughout its duration is described as its envelope. How you shape the envelope of a sound will be covered in more detail later in the BASIC programming strand. But for now all you need to know is that there are four parameters—attack, decay, sustain, release.

The attack of a note describes the rate at which it climbs to its peak volume. And the decay describes how fast its volume falls after the peak volume. The mid-range volume is known as the sustain. And the rate at which the volume falls from the mid-range volume is known as the release.

These terms can easily be understood if you think about how a note is played on a piano. The attack is controlled by how you hit the key. If you hit it hard, quickly, the attack is very sharp. But if the key is stroked the attack is shallower.

After the initial strike of the hammer against the string, the volume decays quite



quickly. But as long that the key is held down—and the peddles are not depressed—the damper will be held back from the string and the note will sustain. It dies away completely when the damper rests back on the string when the key is released.

Attack, decay, sustain and release all have values between 0 and 15. The attack value is stored in the high nybble of memory location D405 with the decay value is in the low nybble. And the values for sustain and release are in the high and low nybbles, respectively, of memory location D406.

Attack, decay and release are all set to zero, while sustain is set to 15.

SETTING THE PITCH

The pitch is controlled by memory locations D400 and D401. And the details of the pitch off each note in the opening phrase of Greensleeves is supplied in a data table at 4360 hex.

The index register Y is loaded with zero and the first byte of the data table is loaded into the high-byte of the pitch control location. Y is incremented and the next data byte is loaded into the low byte of the pitch control location and Y is incremented again. The index is then saved in FC.

TIMING THE NOTE

The length of the note also has to be specified. But there is no location in the SID chip which controls it. You have to time it and switch the note off when it sounds long enough, yourself.

So first of all you have to reset the real time clock. This is controlled by the SETTIM routine in the Kernal ROM. Zero is loaded into the accumulator and transferred into X and Y. All three of these registers have to be zero to reset the clock. The processor then jumps to SETTIM which begins at FFDB.

How long the note should be held for is kept in the third item in the data table. So the index is restored from FC. Then the next data byte is loaded into the accumulator. Then the Y index is incremented and saved back in FC and the length of the note is stored in FD.

Memory location D404 is the SID chip's control register and each bit of it controls a function. Here 17 (11 hex) is loaded into it—17 is 00010001 in binary so bit zero and bit four are set.

Bit zero is the *gate*—setting it turns the sound on. And bit four gives a sound with a triangular waveform which has a mellow, flute-like quality. What the other bits do will be explained in a later article on music and sound effects.

Once the sound has been set off the processor jumps to the RDTIM subroutine in the Kernal ROM which reads the real time clock. The clock counts in 1/60ths of a second and stores the time in three bytes. Despite what the Programmer's Reference Guide says, when you read the time the least significant is returned in the accumulator, the next most in the X register and most significant in Y.

Musical notes only last for a very short time so it is the byte in the accumulator that is compared to the duration parameter in FD. If it is not equal—in other words the clock has not reached the value of the parameter set yet—the BNE instruction branches back to the JSR instruction which goes off and reads the time again. But when the time is up the BNE condition is not fulfilled and the processor moves on.

The Y index is restored from FC again. Zero is loaded into the accumulator and stored in D404, the SID control register. This switches the sound off again.

The note counter in FB is then decremented, and if it has not counted down to zero the BNE instruction loops back to deal with the next note. If it has counted down to zero, the contents of the accumulator—which are still zero—are stored in D418 which switches the volume off. Then the processor returns to BASIC because this is the end of part three of Cliffhanger.

THE DATA

The assembly language routine above can be used to play any tune, providing you change the value of the note counter stored in FB. The shape of the envelope and the type of sound produced can be changed by making suitable adjustments to the values put into D404, D405 and D406.

But it won't play anything at all unless you give it some data. Again there is no point in doing this in machine code and the following BASIC program POKEs the data for Greensleeves into a table which the machine code program can then access.

```
10 ADD=17248:FOR I=0TO32000
```



```
20 READ A%:POKE ADD+I,A%
30 IF A%=0 GOTO 50
40 NEXT
50 END
100 DATA 28,126,16,45,198,32
200 DATA 51,97,16,57,172,64
300 DATA 61,126,16,57,172,32
400 DATA 51,97,32,43,52,16
500 DATA 34,75,64,38,126,16
600 DATA 43,52,32
700 DATA 45,198,32,38,126,16
800 DATA 38,126,64,34,75,16
900 DATA 38,126,32
1000 DATA 43,52,32,36,85,16
1100 DATA 28,214,32,0
```

Although the tune is played by a machine code routine, it needs data for the tune. Again there is no point in entering the data as machine code. So there is a BASIC program that POKEs the data into a data table in the protected part of memory where the assembly language program which follows can access it.

Note that this program uses two voices, so it will not work on the Electron. The Electron's Cliffhanger will have to remain, sadly, silent though the rest of the game will work.

Don't forget to press BREAK and type PAGE=&3000 and NEW again before you key in the program.

```
30 REM ////////////////////
40 REM / Data for voice two /
50 REM ////////////////////
60 DATA137,10
70 DATA149,148
80 DATA157,10
90 DATA165,15
100 DATA169,5
110 DATA165,10
120 DATA157,148
130 DATA145,10
140 DATA129,15
150 DATA137,5,1886
160 DATA145,10
170 DATA149,148
180 DATA137,10
190 DATA137,15
200 DATA133,5
210 DATA137,10
220 DATA145,148
230 DATA133,10
240 DATA117,20
250 DATA137,10,1756
260 DATA149,148
270 DATA157,10
280 DATA165,15
290 DATA169,5
300 DATA165,10
310 DATA157,148
320 DATA145,10
330 DATA129,15
340 DATA137,5
350 DATA145,10,1894
360 DATA149,143
370 DATA145,5
380 DATA137,10
390 DATA133,15
400 DATA125,5
410 DATA133,10
420 DATA137,158
430 DATA137,20
440 DATA177,158
450 DATA177,15,1989
460 DATA169,5
470 DATA165,10
480 DATA157,148
490 DATA145,10
500 DATA129,15
510 DATA137,5
520 DATA145,10
530 DATA149,148
540 DATA137,10
550 DATA137,15,1846
560 DATA133,5
570 DATA137,10
580 DATA145,148
590 DATA133,10
600 DATA117,30
610 DATA177,158
620 DATA177,15
630 DATA169,5
640 DATA165,10
650 DATA157,148,2049
660 DATA145,10
670 DATA129,15
680 DATA137,5
690 DATA145,10
700 DATA149,143
710 DATA145,5
720 DATA137,10
730 DATA133,15
740 DATA125,5
750 DATA133,10,1606
760 DATA137,158
770 DATA137,30
773 REM ////////////////////
775 REM / Data for voice three /
778 REM ////////////////////
780 DATA137,158
790 DATA149,30
800 DATA129,158
810 DATA145,30
820 DATA137,158
830 DATA121,30
840 DATA117,138
850 DATA165,10,2274
860 DATA145,10
870 DATA117,30
880 DATA137,158
890 DATA149,30
900 DATA129,158
910 DATA145,30
920 DATA137,143
930 DATA145,5
940 DATA149,5
950 DATA157,5,1984
960 DATA165,20
970 DATA117,10
980 DATA137,138
990 DATA185,10
1000 DATA165,10
1010 DATA137,20
1020 DATA149,143
1030 DATA157,5
1040 DATA165,10
1050 DATA165,15,1923
1060 DATA157,5
1070 DATA149,10
1080 DATA129,143
1090 DATA137,5
1100 DATA145,10
1110 DATA145,15
1120 DATA137,5
1130 DATA129,10
1140 DATA137,143
1150 DATA145,5,1761
1160 DATA149,10
1170 DATA149,15
1180 DATA145,5
1190 DATA137,10
1200 DATA117,138
1210 DATA165,10
1220 DATA145,10
1230 DATA117,20
1240 DATA109,10
1250 DATA101,143,1705
1260 DATA109,5
1270 DATA117,10
1280 DATA117,15
1290 DATA109,5
1300 DATA101,10
1310 DATA129,143
1320 DATA137,5
1330 DATA145,10
1340 DATA145,15
1350 DATA137,5,1469
1360 DATA129,10
1370 DATA137,143
1380 DATA145,5
1390 DATA149,5
1400 DATA157,5
1410 DATA165,20
1420 DATA117,10
1430 DATA137,138
1440 DATA185,10
1450 DATA165,10,1842
1460 DATA137,30
1470 DATA26151
1480 REM //////////////
1490 REM / Read in data /
```

```
1500 REM ///////////////
1510 S%=Ø
1520 FORA%=ØTO13
1530 T%=Ø
1540 FORB%=ØTO9
1550 READ C%,D%
1560 ?(&FD7+A%*2Ø+B%*2)=C%
1570 ?(&FD8+A%*2Ø+B%*2)=D%
1580 T%=T%+C%+D%
1590 NEXT
1600 READC%
1610 IFC%< >T% PRINT"Error in
   lines□";A%*100+6Ø;"–";A%
   *100+15Ø:END
1620 S%=S%+T%
1630 NEXT
1640 READC%,D%
1650 ?&1ØEF=C%
1660 ?&1ØFØ=D%
1670 S%=S%+C%+D%
1680 READ C%
1690 IFC%< >S% PRINT"Error in data":END
1700 Space=&1ØF1
1710 REM ////////////////////////
1720 REM / Assemble machine code /
1730 REM ////////////////////////
1740 FORPASS=ØTO3STEP3
1750 P%=&11ØØ
1760 [OPTPASS
1770 .Music
1780 LDA&8Ø
1790 AND#&8Ø
1800 BEQLb1
1810 RTS
1820 .Lb1
1830 LDA&8Ø
1840 AND#&4
1850 BEQLb2
1860 LDA&8Ø
1870 AND#&FB
1880 ORA#3
1890 STA&8Ø
1900 LDA#Ø
1910 STA&81
1920 STA&82
1930 .Lb2
1940 LDA&8Ø
1950 AND#&1
1960 BEQLb3
1970 LDA#&8Ø
1980 LDX#&F9
1990 LDY#&FF
2000 JSR&FFF4
2010 TXA
2020 BEQLb3
2030 LDA#2
2040 STASpace
2050 STASpace+2
2060 LDA#Ø
2070 STASpace+1
2080 STASpace+3
2090 STASpace+5
2100 STASpace+7
2110 LDX&81
2120 LDA&FD7,X
2130 STASpace+4
2140 LDA&FD8,X
2150 AND#128
2160 BEQLb4
2170 LDA#1
2180 STASpace+1
2190 .Lb4
2200 LDA&FD8,X
2210 AND#127
2220 STASpace+6
2230 LDA#7
2240 LDX#Space MOD256
2250 LDY#Space DIV256
2260 JSR&FFF1
2270 LDX&81
2280 INX
2290 INX
2300 STX&81
2310 CPX#144
2320 BNELb3
2330 LDA&8Ø
2340 AND#&FE
2350 STA&8Ø
2360 .Lb3
2370 LDA&8Ø
2380 AND#&2
2390 BEQLb5
2400 LDA#&8Ø
2410 LDX#&F8
2420 LDY#&FF
2430 JSR&FFF4
2440 TXA
2450 BEQLb5
2460 LDA#3
2470 STASpace
2480 STASpace+2
2490 LDA#Ø
2500 STASpace+1
2510 STASpace+3
2520 STASpace+5
2530 STASpace+7
2540 LDX&82
2550 LDA&1Ø67,X
2560 STASpace+4
2570 LDA&1Ø68,X
2580 AND#128
2590 BEQLb6
2600 LDA#1
2610 STASpace+1
2620 .Lb6
2630 LDA&1Ø68,X
2640 AND#127
2650 STASpace+6
2660 LDA#7
2670 LDX#Space MOD256
2680 LDY#Space DIV256
2690 JSR&FFF1
2700 LDX&82
2710 INX
2720 INX
2730 STX&82
2740 CPX#138
2750 BNELb5
2760 LDA&8Ø
2770 AND#&FD
2780 STA&8Ø
2790 .Lb5
2800 RTS
2810 ]NEXT
```

To execute this program to test it type RUN then key in:

```
ENVELOPE 2,1,Ø,Ø,Ø,Ø,Ø,Ø,126,–1,–1,
–1,126,126
ENVELOPE 3,1,Ø,Ø,Ø,Ø,Ø,Ø,126,–1,–1,
–1,126,126
?&8Ø=4:REPEAT:CALL &11ØØ:UNTIL?&8Ø=Ø
```

When Cliffhanger is complete these commands will be given in a bootstrap routine which sets the rest of it running. For now, though, you have to do it by hand.

The ENVELOPE commands above shape the sounds given by channels two and three. As they remain constant throughout the program there is no point in setting them in machine code. Their parameters are simply put into a table where they are referred to when required and that might as well be done in BASIC.

The music is going to play while the rest of the program is running, so it interrupts the main program and executes a little bit. So when you test it you have to CALL it REPEATedly. &8Ø is the byte which controls the tune. When bit 2 is set—by POKEing 4 into it—the tune plays. When the machine code routine puts Ø in this location, the tune stops.

THE DATA

The data for each note is on a separate line. The first figure specifies the pitch of the note, the second the length it is played for. And the third figure, if there is one, is a checksum.

There is a further checksum total at the end of the program. The BASIC program that checks the totals, READs the and POKEs it into the data table works in exactly the same way at the POKEr program in part one of Cliffhanger.

TUNING UP

The memory locations from 1ØF1 to 1ØF8 are used to store the parameters that control the sound output. These are the same as the parameters used in the BASIC SOUND and ENVELOPE commands. So the base address of the table is given a label, Space, so that they can be referred to easily in the program.

Memory location &81 is the data table pointer for voice two and &82 performs the same function for voice three. &80 is the control byte switching the tune on and off.

In Line 1780 the contents of the control byte are loaded into the accumulator. These are then ANDed with 80 hex—10000000 binary—to check whether bit seven is set. If it is set, the ANDing gives the result 1, so the zero flag is not set and the condition of the BEQ on Line 1800 is not fulfilled. The branch is not made and the processor hits the RTS and returns. Setting bit seven of &80 is used to switch the tune off—a facility to switch the tune off while you're playing the game will be given later in the program. It does it, of course, by setting bit seven.

If bit seven isn't set the ANDing will give a zero result. So the zero flag will be set and the BEQ will skip the RTS and go straight to the label .Lb1 in Line 1820.

The instruction in Line 1840 checks whether bit two of &80 is set. This is the bit that is set to start playing the tune.

If it is set the contents of &80 are ANDed with &FB and ORed with &3. This clears bit two and sets bits one and zero. Then zero is stored in &81 and &82, to move them to the beginning of the data table. This sets the routine up ready to play the tune.

If bit two is not set the tune is already playing, so it need not be initialized and the instruction in Line 1850 jumps over this section. Bits one and bit zero set indicate that voice two and three are on.

THE VOICES

Lines 1930 to 2350 are almost an exact repeat of Lines 2360 to 2780. The second block deals with channel three while the first deals with channel two. But they both work in exactly the same way.

The first thing that happens in these routines is that the processor checks whether that voice is still playing. As the channels are independent, one can be playing while the other is not. So the instructions in Lines 1950 and 2380 check whether bit zero and bit one, respectively, are set.

The next thing to do is to check if there is space in the sound channel buffer. This is done by making an OSBYTE call. The OSBYTE vector is in &FFF4, so the appropriate parameters are loaded into A, X and Y—exactly as you do in BASIC. (OSBYTE, OSWORD and other operation system calls will be explained fully in a forthcoming article.)

SOUNDING OUT

As voice two is going to be used the number 2 is stored in Space. And as envelope two is going to be used as well, 2 is stored in Space + 2.

Then 0 is stored in Space + 1, Space + 3, Space + 5 and Space + 7. These are the high bytes of the channel, amplitude, pitch and duration parameters.

X is then loaded with the data pointer and the accumulator is loaded with the contents of &FD7 offset by the contents of X. So a byte of the data table is loaded up and stored in Space + 4. This is the low byte of the pitch parameter.

Then the next byte of the data table is loaded up. This is ANDed with 128 to check whether the high bit is set. Setting the high bit of the second of each pair of bytes in the data table indicates that you've reached the beginning of a bar. If it is set, Space + 1 is then set to 1, this synchronizes the two channels on the sound chip.

Whether the tune is at the beginning of a bar or not the next byte is loaded up again and ANDed with 127. This clears the high byte—resets to 0 whether it was set or not—and leaves the rest of the bits alone. The result is stored in Space + 6 which is the low byte of the duration parameter.

The OSWORD routine whose vector is at &FFF1 is called. But beforehand the accumulator is set to seven—which turns the OSWORD routine into the equivalent of the BASIC SOUND command. X and Y are loaded with the low and high bytes of the pointer respectively. And it is the JSR&FFF1 that actually makes the sound.

THE NEXT NOTE

X is loaded with the contents of &81, incremented twice and stored back in &81. This moves the data pointer onto the beginning of the next pair.

Then X is compared with 144. There are 144 bytes in the data table for channel two, so when X have rached 144 the last note has been played and the tune is over.

The contents of &80 are then loaded up and ANDed with &FE. This clears bit zero to zero, telling the routine that the tune is over as far as voice two is concerned.

Then the same steps are gone through all over again for channel three. And when it has played its note. The processor returns to BASIC, ready to be called again.

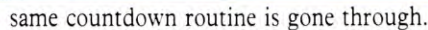
The Dragon plays Greensleeves if you execute the following program:

```
        ORG     30000
        LDX     #MUSIC
        STX     MUPOINT
        LDA     #19
        PSHS    A
        LDA     $FF01
        ANDA    #247
        STA     $FF01
        LDA     $FF03
        ANDA    #247
        STA     $FF03
        LDA     $FF23
        ORA     #8
        STA     $FF23
MAIN    LDU     MUPOINT
        ORCC    #$50
        PULU    A,X
        CMPU    #MUSIC + 57
        BLO     MONE
        LDU     #MUSIC
MONE    STU     MUPOINT
        PSHS    X
        LDB     #252
MTWO    STB     $FF20
MTHR    LEAX    -1,X
        BNE     MTHR
        LDX     ,S
        CLR     $FF20
MFOU    LEAX    -1,X
        BNE     MFOU
        LDX     ,S
        DECA
        BNE     MTWO
        LEAS    2,S
        DEC     ,S
        BNE     MAIN
        ANDCC   #$AF
        PULS    A,PC
MUPOINT FDB     $ 758A
MUSIC   FCB     98,0,189,233,0,158,
                131,0,141,220,0,125,
                78,0,118,147,0,125,
                255,0,141,110,0,168,
                131,0,212,49
        FCB     0,189,110,0,168,233,
                0,158,98,0,189,147,0,
                189,44,0,212,98,0,
                189,220,0,168,92,0,
                200,220,0,252
```

This program has two entry points. One is at 30000—if you call it there, the whole tune is played. The other is at 30008 which plays a single note. This is used when you want to play the tune when something else is happening. It means that you don't have to stop the action while the whole tune plays. You can play one note, do something else on the screen, say, then play another. This preserves the illusion of smooth action.

SETTING UP THE WHOLE TUNE

The value of the label MUSIC is loaded into the data byte labelled by MUPOINT. Although



when the program is listed here, and when you assemble it, it already has the value of MUSIC in that byte, MUPOINT is actually used to store the position of the last note played. So if you are playing the routine a note at a time there is a record of the last note played.

A is then loaded with 19—which is the number of notes in the first phrase of Greensleeves. Now the whole tune is ready to play.

TAKING NOTES

If you don't want to play the whole tune you load the accumulator with the number of notes you want to play. So if you want to play the tune a note at a time you simply load A with 1.

The first thing that part of the program does is push the value of A onto the hardware stack. This is temporary storage. The program will have to refer to the number of notes it has to play later.

The following nine instructions are a sound enable routine which sets the input/output chip for outputting noise.

THE MAIN ROUTINE

MUPOINT points to the next note to be played, so LDU MUPOINT loads the user stack pointer with the pointer to the next byte of music data. This effectively turns the music data given at the end of the routine into the user stack.

ORCC #$50 disables the interrupts by masking the condition code register. Whatever the value of bits four and six of the condition code register, ORing it with 50 hex—80 decimal—sets them to 1.

As U is pointing to the music data for the next note, PULU A,X pulls the next byte of the music data into A, the next two bytes into X and increments U by three, moving the pointer onto the data for the next note.

The data for the 19 notes comes in threes. The first byte is the number of cycles you want the note to play for. And the next two bytes carry the delay time, which is a measure of the wavelength of the note to be played.

U is then compared to #MUSIC+57, which is the address of the end of the music data. BLO:1 branches if lower, so if the end of the music data has not been reached the next instruction is skipped. This sets the data pointer back to the beginning of the data again so the music will play over again next time.

In either case the value of the pointer, which is in U, is stored back in MUPOINT so that the processor will know which note its on next time. The delay in X is pushed onto the hardware stack for temporary storage.

LDB #252 and STB $FF20 calls the digital/analog converter and sets it up for outputting to the TV speaker. LEAX −1,X decrements X and BNE loops back until X is counted down to zero.

X is then reloaded with the value on the stack with the LDX ,S. This acts like a PULS X only it does not increment the stack pointer. CLR $FF20 turns the TV speaker off and the same countdown routine is gone through.

X is reloaded again and the number of cycle the sound has to play for—which is in A—is decremented. BNE then loops back and plays the next cycle of the note until A has counted down to zero.

Then LEAS 2,S clears the stack, and DEC ,S decrements the contents of the next byte up the hardware stack, which was the number of notes. This, you remember, was pushed onto the hardware stack to start with. BNE MAIN loops back again to play the next note, if another is required.

ANDCC #$AF re-enables the interrupts by unmasking the condition code register. And PULS A,PC pulls the note value—now zero—off the hardware stack for housekeeping purposes and the next two bytes. These are the return address of the subroutine, stored there when the music routine was called. This acts exactly the same as an RTS.

MULTI-KEY CONTROL

If you want to have maximum control when game playing, instead of using a joystick, program your keyboard to detect more than one keypress

One of the most important features of any game is the quality of the interaction between the user and the computer. For most purposes, a joystick offers more sophisticated control, but the keyboard is probably more versatile. To test which key is being pressed, the keywords INKEY$ or GET$ are normally used.

INKEY$ and GET$ can detect most characters, but their major disadvantage is that they can detect two simultaneous keypresses only if one key is SHIFT—or a similar control key such as CTRL on the Acorns and Commodores, SYMBOL SHIFT on the Spectrum.

Detecting such combinations is adequate for most purposes, but a problem arises when, for example, you wish to control the vertical and horizontal movement, fire the laser and drop the smart bombs of a craft. Altogether, such actions require a total of four simultaneous keypresses. The best way to solve the problem—although in fact it may not always be possible to solve it—depends on the make of computer.

THE EFFECT OF PRESSING KEYS

To understand how to detect more than one keypress, it is important to know how the computer works out which key has been pressed. Two methods are in popular use on home micros. In the first, the computer *polls* the keyboard—it scans the keys at regular intervals to see if one has been pressed. The Dragon, for example, carries out a polling every hundredth of a second, and the Commodores also work on this principle.

In the second method, the depression of a key causes a message—called an interrupt—to be sent to the processor. This action causes the computer to stop what it is doing briefly to tend to the keyboard. The task then is to find out which key has been pressed. This method is much more efficient and versatile than polling, because if the keys are idle, there is no need to tend them, but if a key is being held down, you want the strongest signals from it to register.

Whatever method of detecting keypresses is used, the computer must be able to identify any key easily and quickly. Most keyboards use a system called matrix generation, by which each key generates a number according to its position in a matrix. What happens when this number is generated depends on how the computer was designed.

S

The Spectrum keyboard is arranged as four rows of ten keys. But for the computer's purposes, imagine each row is halved, so there are eight groups of five keys. Each group communicates with an input/output line or I/O port. The Spectrum keyboard is particularly suited to multiple key detection, because there are 65536 of these ports, each identified by an address between Ø and 65535. The table below gives the port address for each of the eight groups.

KEY GROUP	n	PORT
CAPS/SHIFT Z X C V	Ø	65276
A S D F G	1	65Ø22
Q W E R T	2	6451Ø
1 2 3 4 5	3	63486
6 7 8 9 Ø	4	61438
Y U I O P	5	57342
H J K L ENTER	6	4915Ø
B N M SYM/SHIFT SPACE	7	32766

To work out the address of each group, use the formula: $254+256*(255-2^n)$ where n is the key group listed in the table. Alternatively, enter and RUN this short program to calculate the addresses:

```
1Ø INPUT "ENTER KEY GROUP NUMBER
   :□";n
2Ø PRINT "KEY GROUP NUMBER□";n
3Ø PRINT "PORT ADDRESS□";
   254+256*(255-2↑n)
4Ø GOTO 1Ø
```

Each of the port addresses in the table includes a number that depends on the key being pressed. If you have a printer, or any sort of interface connected, you should remove this or you may get spurious results. Replace the first program with these two lines, then RUN them and play about with keys 1 to 5:

```
1Ø PRINT AT Ø,Ø;IN 63486
2Ø GOTO 1Ø
```

Notice how the numbers on the screen change when different combinations of keys are pressed.

The value in the port address is held in one byte, of which the eighth and sixth bits are always 1. Usually, the seventh bit is Ø, but it changes to 1 when there is a signal at the EAR socket and when the computer becomes warm. Bits 1 to 5 represent the keys in each group. Normally, they are 1, but when a key is pressed, the bit corresponding to the key changes to Ø. The fifth bit corresponds to the key nearest the middle of the row, and the first bit for the key at the edge of the row.



- THE EFFECT OF PRESSING KEYS
- UNDERSTANDING HOW YOUR KEYBOARD WORKS
- THE DIFFERENCE IN METHODS
- IMPLEMENTING GAME CONTROLS
- MATRIX GENERATION
- MULTIPLE KEY DETECTION
- DETECTING SIMULTANEOUS KEYPRESSES
- AUTO-REPEAT FACILITY

When each of the eight bits is 1, it contributes to the value in the port, but contributes nothing when it is Ø. The value contributed by each bit is as follows:

Bit	8	7	6	5	4	3	2	1
Amount contributed	128	64	32	16	8	4	2	1

So if the seventh bit is Ø and 5 is pressed, the bit pattern in port 63486 is 1Ø1Ø1111, and the value in this port will be 128+Ø+32+Ø+8+4+2+1 =175. In this manner, you can detect simultaneous keypresses of the five keys in the group.

This applies to each of the other groups of keys as well and you can add lines to the program above to experiment with printing the values in their port addresses.



[C= logo]

To see the matrix numbers generated by keypresses on these machines, enter the following program:

```
20 PRINT "[CLR]MATRIX NUMBER
   IS□";PEEK (197)
30 GOTO 20
```

RUN the program and notice that each key you press generates a different number. These numbers are not changed, however, even if you press a key simultaneously with SHIFT or C=.

The number of the key is stored at location 197 (as shown at Line 2Ø), but it is then translated, using a table in memory, into its ASCII code. This code is placed in the keyboard buffer, to be acted on by the processor.

Normal keys can have different meanings if they are pressed at the same time as the control keys, such as SHIFT. So other translation tables are needed to decode such double presses.

Pressing SHIFT, CTRL or C= causes a flag to be set, so the correct table can be accessed. There is, however, no conversion table to decode the number in location 197 when normal keys are pressed simultaneously.

Add the next line to the program above to see the status of the SHIFT flag, which is stored at location 653:

```
25 PRINT "SHIFT FLAG VALUE ";
   PEEK(653)
```

To overcome the limitation of being able to detect only certain simultaneous keypresses, you can restrict the number of key codes that can be stored in the keyboard buffer. Normally, up to ten characters can be stored, but there are instances when fewer keys in the buffer would allow the Operating System to ignore some keypresses and detect only the most recent ones.

You can reduce the capacity of the buffer by POKEing location 649 (which normally has a value of 1Ø) with a smaller number. Beware, however, of replacing the 1Ø with larger numbers, because you might disrupt the keyboard and screen handling routines near this address.

It can also be an advantage not to merely reduce the capacity of the buffer, but to empty it before detecting a key. You can achieve this by modifying the loop to read or get a

character from the keyboard:

```
10 GET A$:IF A$=""THEN 10
```

It is useful to include such a line in a program that relies on accurate input from the user.

As well as being able to empty the buffer, you will find it useful to let the keys auto-repeat. Enter POKE 650,128 as a direct command, then hold down any of the normal keys. You should see the character printed repeatedly on the screen. Imagine this character is detected and so causes a burst from a laser cannon, for example. The auto-repeat facility will give not just a burst, but a barrage of laser fire, making an exciting attack—albeit with one keypress. To switch off the repeat function, enter POKE650,0.



Acorn computers convert the matrix numbers of different keys into ASCII codes, then place these in a 32-character buffer. As with the other micros, they let you detect a single keypress, or those with SHIFT, by using the BASIC keyword INKEY or INKEY$. As with the Commodores, there is no simple means of detecting when two normal keys are pressed simultaneously, but you can simulate this condition closely by using INKEY followed by a negative number in brackets. The negative number for each key is different from the ASCII code, and there is a table in the User Guide listing the numbers.

At the instant the command is executed, the Operating System causes the keyboard to be scanned and can identify a keypress, irrespective of whether there are other characters in the buffer waiting to be acted on. The essential point about this facility is that it is instant, so it is an immediate method of branching a program.

The keys on the Dragon keyboard are arranged in a matrix of seven rows by eight columns. Nine bytes of memory are used to store the state of a row (the first byte) and of the eight columns (the remaining eight bytes). The arrangement of the matrix and the method of decoding the state of each row make it impossible normally to detect more than single keypress at any given moment.

You can, however, fool the keyboard scanner into accepting that all the keys in a row are released, when its next scan will record any key being pressed as a new occurrence. This involves poking the value &-FF at the first byte. This action has the effect of recording the state of the rows.

This method is not really satisfactory, because it requires such changes to be made frequently in the program. A better method is to call a machine code routine that forces a complete keyboard scan when required. This is the method used in the programs at the end of this article.






On all the computers except the Commodores, detecting, say, six keypresses at once is not much harder than detecting two. Although the Spectrum is the only machine which genuinely reads all the keys at once, you can usually simulate simultaneous detection satisfactorily on the others. On the Commodores, the only way in which you can do this, however, is if you are prepared to use the SHIFT, C= , and CTRL keys, together with one normal key—giving you a maximum of four keys that can be detected at the same time. Enter the next program to see the degree of success that can be achieved:

```
10 BORDER 0:PAPER 0:INK 7
20 CLS
30 LET y=11:LET x=15
40 LET p=63486
50 GOSUB 220
60 IF i=175 THEN GOSUB 290
70 LET p=61438
80 GOSUB 220
90 IF i=187 THEN GOSUB 380
100 IF i=175 THEN GOSUB 320
110 IF i=183 THEN GOSUB 350
120 IF i=189 THEN GOSUB 410
130 IF i=185 THEN GOSUB 380:
    GOSUB 410
140 IF i=173 THEN GOSUB 320:
    GOSUB 410
150 IF i=181 THEN GOSUB 350:
    GOSUB 410
160 IF i=171 THEN GOSUB 380:
    GOSUB 320
170 IF i=179 THEN GOSUB 380:
    GOSUB 350
180 IF i=169 THEN GOSUB 380:
    GOSUB 320: GOSUB 410
190 IF i=177 THEN GOSUB 380:
    GOSUB 350: GOSUB 410
200 GOSUB 250
210 GOTO 40
220 LET i=IN p
230 IF i>191 THEN LET i=i-64
240 RETURN
250 PRINT AT y,x;"□+□"
260 PRINT AT y+1,x;"□□□"
270 PRINT AT y-1,x;"□□□"
280 RETURN
290 IF x<1 THEN RETURN
300 LET x=x-1
310 RETURN
320 IF y>19 THEN RETURN
330 LET y=y+1
340 RETURN
350 IF y<2 THEN RETURN
360 LET y=y-1
370 RETURN
380 IF x>28 THEN RETURN
390 LET x=x+1
400 RETURN
410 BEEP .004,20
420 BEEP .004,10
430 PRINT AT y,x;"□□□"
440 RETURN
```



```
10 POKE 650,128:PRINT "□";
    CHR$(8)
100 P=1024:C=55296:SP=40:K1=
    37:K2=36:AX=40:AY=25:Y=12:
    X=20:POKE 53281,0
110 XX=0:YY=0:A=PEEK(197):IF A=K1
    THEN YY=-1
120 IF A=K2 THEN YY=1
130 IF PEEK(653)=2 THEN XX=-1
140 IF PEEK(653)=1 THEN XX=1
150 IF PEEK(653)>3 THEN GOSUB 300
160 POKE P+(Y*SP)+X,46:POKE
    C+(Y*SP)+X,RND(1)*6+1
200 IF X+XX>-1 AND X+XX<AX THEN
    X=X+XX
210 IF Y+YY>-1 AND Y+YY<AY THEN
    Y=Y+YY
220 POKE P+(Y*SP)+X,87:GOTO 110
300 VV=54296:WW=VV-20:
    AA=WW+1:HH=WW-3
310 POKE VV,15:POKE WW,33:POKE AA,180
320 FOR Z=1 TO 15 STEP.3:POKE
    53280,Z:POKE HH,34+Z:NEXT Z
330 POKE HH,0:POKE WW,0:RETURN
```



Delete Lines 320 and 330 of the Commodore 64 program, then replace Lines 100, 300 and 310 with the following:

```
100 P=7680:C=38400:SP=22:K1=
    44:K2=36:AX=22:AY=23:Y=10:
    X=11:POKE 36879,14
300 POKE 36878,15
310 FOR Z=8 TO 15 STEP.1:POKE
    36879,Z:POKE 36877,200-
    (Z*5):NEXT Z:RETURN
```



```
10 MODE1:COLOUR 3:COLOUR 129
20 CLS:VDU 23;8202;0;0;0;
30 X=19:Y=16
40 REPEAT
50 IF INKEY(-58) THEN PROCup
60 IF INKEY(-42) THEN PROCdown
70 IF INKEY(-26) THEN PROCleft
```

```
80 IF INKEY(−122) THEN PROCright
90 IF INKEY(−99) THEN PROCfire
100 PROCprint
110 UNTIL FALSE
120 DEFPROCup
130 IF Y<1 THEN ENDPROC
140 Y=Y−1
150 ENDPROC
160 DEFPROCdown
170 IF Y>28 THEN ENDPROC
180 Y=Y+1
190 ENDPROC
200 DEFPROCleft
210 IF X<1 THEN ENDPROC
220 X=X−1
230 ENDPROC
240 DEFPROCright
250 IF X>36 THEN ENDPROC
260 X=X+1
270 ENDPROC
280 DEFPROCfire
290 SOUND &10,−10,6,1
300 PRINT TAB(X,Y);"□□□";
310 ENDPROC
320 DEFPROCprint
330 PRINT TAB(X,Y);"□+□";
340 PRINT TAB(X,Y+1);"□□□";
350 PRINT TAB(X,Y−1);"□□□";
360 ENDPROC
```

```
10 CLEAR200,32746
20 FORK=32747TO32767:READA:
   POKEK,A:NEXT
30 DATA 48,140,9,191,1,155,134,126,
   183,1,154,57,52,3,134,127,183,1,81,
   53,131
40 V=223:P=1300:L=1300:CLS3
50 IF PEEK(341)=V AND P>1055 THEN
   P=P−32
60 IF PEEK(342)=V AND P<1504 THEN
   P=P+32
70 IF PEEK(343)=V AND P>1024 THEN
   P=P−1
80 IF PEEK(344)=V AND P<1535 THEN
   P=P+1
90 IF PEEK(345)=V THEN SOUND200,1
100 POKE L,175:POKE P,43:L=P:
   GOTO50
```

The Dragon program uses a short machine code routine, so Dragon users should key the listing above and RUN it. Once you are sure there are no errors, enter the next line, then the program will be complete:

```
35 EXEC32747
```

RUN the program to see a target printed at the centre of the screen. You can use the arrow keys to move the target in any direction about the screen, and press 9, CAPS SHIFT on the Acorns, or the space bar on the Dragon to fire. The Commodore keys are chosen carefully, because normal keys cannot be detected when they are pressed together. Since you never need to move up and down at the same time, K and M are used for these actions. C= and SHIFT are used for left and right movement, and CTRL for fire.

The structure of the programs is different in most cases, but they use the methods described above to detect five keys—not all simultaneously in the case of the Commodores, but sufficiently close to give a fair simulation. Press the keys to move the target up and left, for example, and it will move diagonally—and you can fire while doing so.

The principle is taken a step further in the next program, which detects six keys to make possible a game for two players. There is no listing for the Commodores, because you cannot detect six keys simultaneously on those micros, so the game would not be fair to both players.

```
10 BORDER Ø:PAPER Ø:INK 7
20 BRIGHT Ø:OVER Ø:CLS
30 PRINT AT 1,7;INK 6;FLASH 1;
   "□G□U□N□F□I□G□H□T!□"
40 PRINT:PRINT
50 PRINT INK 5;"□□WIN POINTS BY
   SHOOTING YOUR□□□OPPONENT.
   EACH PLAYER HAS SIX□□□BULLETS."
60 PRINT:PRINT
70 PRINTTAB 7; INVERSE 1;
   "C□O□N□T□R□O□L□S:□"
80 PRINT:PRINT "□PLAYER 1
   □□□□□□□□□□□□□□
   □□PLAYER 2□"
90 PRINT:PRINT "□□□□1□□□□□
   ———UP———□□□□□□Ø□□
   □□"
100 PRINT:PRINT "□□□□Q□□□
   □□——DOWN——□□□□□P
   □□□"
110 PRINT:PRINT "□□□□A□□□
   □□——FIRE——□□□□□
   ENTER"
120 PRINT:PRINT:PRINT TAB 6;"MAY THE
   BEST MAN WIN"
130 FOR n=USR "a" TO USR "i"+7
140 READ d
150 POKE n,d
160 NEXT n
170 DATA 1,3,6,15,31,63,15,15
180 DATA 192,192,Ø,192,192,192,Ø,135
190 DATA 15,15,15,15,15,12,12,14
200 DATA 248,248,128,128,128,192,192,224
210 DATA Ø,Ø,Ø,Ø,Ø,Ø,126,Ø
220 DATA 3,3,Ø,3,3,3,Ø,113
230 DATA 128,192,96,240,248,252,240,240
240 DATA 15,15,1,1,1,3,3,7
250 DATA 240,240,240,240,240,48,48,112
260 LET y1=1Ø: LET y2=1Ø
270 LET b1=6: LET b2=6
280 LET s1=Ø: LET s2=Ø
290 RESTORE 33Ø: FOR n=1 TO 8
300 READ d,p
310 BEEP d,p
320 NEXT n
330 DATA .1,7,.Ø9,12,.1,7,.Ø9,12,.6,
   7,.45,2,.45,6,.5,Ø
340 PRINT #1;AT Ø,Ø; FLASH 1;
   "□□□PRESS ANY KEY TO PLAY
   GAME□□□"
350 LET p=254: GOSUB 57Ø
360 IF i=191 THEN GOTO 35Ø
370 INK 4: BRIGHT 1: CLS
380 PRINT INVERSE 1;"PLAYER 1□Ø□□□
   □□□□□□□□□□ØPLAYER 2"
390 PRINT "BULLETS:□6□□□□□
   □□□□□□□□6□:BULLETS"
400 PRINT #1;AT Ø,Ø; INVERSE 1;
410 GOSUB 6ØØ
420 LET p=63486: GOSUB 57Ø
430 IF i=19Ø THEN GOSUB 81Ø
440 LET p=61438: GOSUB 57Ø
450 IF i=19Ø THEN GOSUB 84Ø
460 LET p=6451Ø: GOSUB 57Ø
470 IF i=19Ø THEN GOSUB 87Ø
480 LET p=57342: GOSUB 57Ø
490 IF i=19Ø THEN GOSUB 9ØØ
500 LET p=4915Ø: GOSUB 57Ø
510 IF i=19Ø THEN GOSUB 1Ø2Ø
520 LET p=65Ø22: GOSUB 57Ø
530 IF i=19Ø THEN GOSUB 93Ø
540 GOSUB 6ØØ
550 IF b1=Ø AND b2=Ø THEN GOTO 111Ø
560 GOTO 42Ø
570 LET i=IN p
580 IF i>191 THEN LET i=i−64
590 RETURN
600 PRINT AT y1,1;CHR$144;CHR$145
610 PRINT AT y1+1,1;CHR$146;CHR$147
620 PRINT AT y2,29;CHR$149;CHR$15Ø
630 PRINT AT y2+1,29;CHR$151;CHR$152
640 PRINT AT y1−1,1;"□□"
650 PRINT AT y1+2,1;"□□"
660 PRINT AT y2−1,29;"□□"
670 PRINT AT y2+2,29;"□□"
680 PRINT AT Ø,9; PAPER 4; INK 9;s1
690 PRINT AT Ø,22; PAPER 4; INK 9;s2
700 PRINT AT 1,9;b1
710 PRINT AT 1,22;b2
720 RETURN
730 PRINT AT 1Ø,1Ø;"AAGH! GOT ME!"
740 RESTORE 78Ø: FOR n=1 TO 11
750 READ d,p
760 BEEP d,p
770 NEXT n
780 DATA .5,2,.4,2,.2,2,.5,2,.3,5,.2,
   4,.4,4,.2,2,.4,2,.2,1,.5,2
```

```
790 PRINT AT 1Ø,1Ø;"□□□□□□□□□□□□□"
800 RETURN
810 IF y1<4 THEN RETURN
820 LET y1=y1-1
830 RETURN
840 IF y2<4 THEN RETURN
850 LET y2=y2-1
860 RETURN
870 IF y1>18 THEN RETURN
880 LET y1=y1+1
890 RETURN
900 IF y2>18 THEN RETURN
910 LET y2=y2+1
920 RETURN
930 IF b1=Ø THEN RETURN
940 BEEP .Ø1,4: BEEP .Ø1,Ø
950 FOR n=3 TO 27
960 PRINT AT y1,n;"□";CHR$148
970 NEXT n
980 PRINT AT y1,27;"□□"
990 IF y1=y2 OR y1=y2+1 THEN LET s1=s1+1: GOSUB 73Ø
1ØØØ LET b1=b1-1
1Ø1Ø RETURN
1Ø2Ø IF b2=Ø THEN RETURN
1Ø3Ø BEEP .Ø1,Ø: BEEP .Ø1, -1Ø
1Ø4Ø FOR n=27 TO 3 STEP -1
1Ø5Ø PRINT AT y2,n;CHR$148;"□"
1Ø6Ø NEXT n
1Ø7Ø PRINT AT y2,3;"□"
1Ø8Ø IF y2=y1 OR y2=y1+1 THEN LET s2=s2+1: GOSUB 73Ø
1Ø9Ø LET b2=b2-1
11ØØ RETURN
111Ø IF s1>s2 THEN PRINT AT 1Ø,8; FLASH 1;"□PLAYER 1 WINS!□"
112Ø IF s2>s1 THEN PRINT AT 1Ø,8; FLASH 1;"□PLAYER 2 WINS!□"
113Ø IF s1=s2 THEN PRINT AT 1Ø,8; FLASH 1;"□GAME IS DRAWN!□"
114Ø GOTO 26Ø
```

```
1Ø MODE1
2Ø VDU 23;82Ø2;Ø;Ø;Ø;
3Ø COLOUR 129:PRINT TAB(15,1);"□Gunfight!□"
4Ø COLOUR 128:PRINT TAB(2,4);"Win points by shooting your opponent."
5Ø PRINTTAB(6,6);"Each player has six bullets."
6Ø COLOUR 2:PRINT TAB(3,8);"Controls:"
7Ø COLOUR 1:PRINT TAB(5,1Ø);"Player□1□□□□□□□□□□□□□□Player□2"
8Ø PRINT TAB(8,12);"Q□□□□□□---UP---□□□□□□@"
9Ø PRINT TAB(8,13);" A□□□□□□--DOWN--□□□□□□;"
1ØØ PRINT TAB(8,14);" Z□□□□□□--FIRE--□□□□□□."
11Ø COLOUR 13Ø:PRINT TAB(8,16);" May the best man win!□"
12Ø VDU 23,24Ø,1,3,6,15,31,63,15,15
13Ø VDU 23,241,192,192,Ø,192,192,192,Ø,135
14Ø VDU 23,242,15,15,15,15,15,12,12,14
15Ø VDU 23,243,248,248,128,128,128,192,192,224
16Ø VDU 23,244,Ø,Ø,Ø,Ø,Ø,Ø,126,Ø
18Ø VDU 23,245,3,3,Ø,3,3,3,Ø,113
19Ø VDU 23,246,128,192,96,24Ø,248,252,24Ø,24Ø
2ØØ VDU 23,247,15,15,1,1,1,3,3,7
21Ø VDU 23,248,24Ø,24Ø,24Ø,24Ø,24Ø,48,48,112
22Ø REPEAT
23Ø y1%=1Ø: y2%=1Ø
24Ø b1%=6: b2%=6
25Ø s1%=Ø: s2%=Ø
26Ø RESTORE 3ØØ:FOR n%=1 TO 8
27Ø READ d%,p%
28Ø SOUND 1,-15,p%,d%
29Ø NEXT n%
3ØØ DATA 2,81,1,1Ø1,2,81,1,1Ø1,12,81,9,65,9,73,1Ø,53
31Ø COLOUR 3:COLOUR 128:PRINTTAB(6,2Ø);"Press any key to play game";
32Ø *FX15,1
33Ø REPEAT UNTIL GET<>Ø
34Ø CLS
35Ø COLOUR 2:PRINT "Player 1:□Ø□□□□□□□□□□□□□□□□□□Player 2:□Ø"
36Ø PRINT "Bullets□:□6□□□□□□□□□□□□□□□□□□□Bullets□:□6"
37Ø PROCprint_men
38Ø REPEAT
39Ø IF INKEY(-17) THEN PROCleft_up
4ØØ IF INKEY(-72) THEN PROCright_up
41Ø IF INKEY(-66) THEN PROCleft_down
42Ø IF INKEY(-88) THEN PROCright_down
43Ø IF INKEY(-98) THEN PROCleft_fire
44Ø IF INKEY(-1Ø4) THEN PROCright_fire
45Ø PROCprint_men
46Ø UNTIL (b1%=Ø AND b2%=Ø)
47Ø PROCgame_over
48Ø UNTIL FALSE
49Ø DEFPROCprint_men
5ØØ COLOUR 3:PRINT TAB(1,y1%);CHR$(24Ø);CHR$(241)
51Ø PRINT TAB(1,y1%+1);CHR$(242);CHR$(243)
52Ø PRINT TAB(37,y2%);CHR$(245);CHR$(246)
53Ø PRINT TAB(37,y2%+1);CHR$(247);CHR$(248)
54Ø PRINT TAB(1,y1%-1);"□□"
55Ø PRINT TAB(1,y1%+2);"□□"
56Ø PRINT TAB(37,y2%-1);"□□"
57Ø PRINT TAB(37,y2%+2);"□□"
58Ø COLOUR 1:PRINT TAB(1Ø,Ø);s1%
59Ø PRINT TAB(37,Ø);s2%
6ØØ PRINT TAB(1Ø,1);b1%
61Ø PRINT TAB(37,1);b2%
62Ø ENDPROC
63Ø DEFPROCplayer_shot
64Ø PRINT TAB(13,1Ø);"Aaagh! Got me!"
65Ø RESTORE 7ØØ:FOR n%=1 TO 11
66Ø READ d%,p%
67Ø SOUND 1,-15,p%,d%
68Ø TIME=Ø:REPEAT UNTIL TIME>(d%*5)+6
69Ø NEXT n%
7ØØ DATA 1Ø,61,8,61,4,61,1Ø,61,6,73,4,69,8,69,4,61,8,61,4,57,1Ø,61
71Ø PRINT TAB(13,1Ø);SPC(15)
72Ø ENDPROC
73Ø DEFPROCleft_up
74Ø IF y1%<4 THEN ENDPROC
75Ø y1%=y1%-1
76Ø ENDPROC
77Ø DEFPROCright_up
78Ø IF y2%<4 THEN ENDPROC
79Ø y2%=y2%-1
8ØØ ENDPROC
81Ø DEFPROCleft_down
82Ø IF y1%>27 THEN ENDPROC
83Ø y1%=y1%+1
84Ø ENDPROC
85Ø DEFPROCright_down
86Ø IF y2%>27 THEN ENDPROC
87Ø y2%=y2%+1
88Ø ENDPROC
89Ø DEFPROCleft_fire
9ØØ IF b1%=Ø THEN ENDPROC
91Ø SOUND Ø,-15,4,1
92Ø FOR n%=3 TO 35
93Ø COLOUR 1:PRINT TAB(n%,y1%);"□";CHR$(244)
94Ø NEXT n%
95Ø PRINT TAB(36,y1%);"□"
96Ø IF y1%=y2% OR y1%=y2%+1 THEN s1%=s1%+1: PROCplayer_shot
97Ø b1%=b1%-1
98Ø ENDPROC
99Ø DEFPROCright_fire
1ØØØ IF b2%=Ø THEN ENDPROC
1Ø1Ø SOUND Ø,-15,4,1
1Ø2Ø FOR n%=35 TO 3 STEP -1
1Ø3Ø COLOUR 1:PRINT TAB(n%,y2%);CHR$(244);"□"
1Ø4Ø NEXT n%
1Ø5Ø PRINT TAB(3,y2%);"□"
1Ø6Ø IF y2%=y1% OR y2%=y1%+1 THEN s2%=s2%+1: PROCplayer_shot
1Ø7Ø b2%=b2%-1
1Ø8Ø ENDPROC
```

```
1Ø9Ø DEFPROCgame_over
11ØØ COLOUR Ø:COLOUR 2
111Ø IF s1%>s2% THEN PRINT TAB
   (14,8);"Player 1 wins!"
112Ø IF s2%>s1% THEN PRINT TAB
   (14,8);"Player 2 wins!"
113Ø IF s2%=s1% THEN PRINT TAB
   (14,8);"Game is drawn!"
114Ø *FX15,1
```

```
1Ø CLS:PMODE4,1:SS=PEEK(186)*256
   +PEEK(187)
2Ø FORK=SS TO SS+48Ø STEP 32
3Ø FORJ=K TO K+3:READA:POKE J,A:
   NEXTJ,K
4Ø DATA 1,192,3,128,3,192,3,192,6,Ø,Ø,
   96,15,192,3,24Ø
5Ø DATA 31,192,3,248,63,192,3,252,15,
   Ø,Ø,24Ø,15,135,113,24Ø
6Ø DATA 15,248,15,24Ø,15,248,15,24Ø,
   15,128,1,24Ø,15,128,1,24Ø
7Ø DATA 15,128,1,24Ø,12,192,3,48,12,
   192,3,48,14,224,7,112
8Ø DIM L(6),R(6),B(6)
9Ø GET(Ø,Ø)-(15,15),L:GET(16,Ø)
   -(31,15),R,G
1ØØ PCLS:PRINT@8,"G□U□N□
   F□I□G□H□T!"
11Ø PRINT@97,"WIN POINTS BY SHOOTING
   YOUR□□□□□OPPONENT. EACH
   PLAYER HAS SIX□□□BULLETS."
12Ø PRINT:PRINTTAB(8);
   "C□O□N□T□R□O□L□S"
13Ø PRINT:PRINT"□PLAYER 1";TAB
   (22);"PLAYER 2"
14Ø PRINT:PRINT"□□□UP□□□
   □□---UP---□□□□□□-"
15Ø PRINT"□□DOWN□□□□--
   DOWN--□□□□□□@"
16Ø PRINT"□□□□Z□□□□□□--
   FIRE--□□□□□□/"
17Ø PRINT@481,"any key to start";
18Ø IF INKEY$="" THEN 18Ø
19Ø X1=16:X2=232:Y1=88:Y2=88:
   B1=6:B2=6:S1=Ø:S2=Ø:PCLS
2ØØ PUT(X1,Y1)-(X1+15,Y1+15),L,PSET:
   PUT(X2,Y2)-(X2+15,Y2+15),R,PSET
21Ø FORK=1TO6:CIRCLE(1Ø*K,2),1,5:
   CIRCLE(255-1Ø*K,2),1,5:NEXT
22Ø SCREEN1,1:A$=INKEY$
23Ø L1=Y1:L2=Y2
24Ø IF PEEK(341)=223 AND Y1>16 THEN
   Y1=Y1-8
25Ø IF PEEK(342)=223 AND Y1<176 THEN
   Y1=Y1+8
26Ø IF PEEK(343)=253 AND Y2>16 THEN
   Y2=Y2-8
27Ø IF PEEK(338)=251 AND Y2<176 THEN
   Y2=Y2+8
28Ø IF B1=Ø AND B2=Ø THEN 36Ø
```



```
29Ø IF L1=Y1 THEN 31Ø
3ØØ PUT(X1,L1)-(X1+15,L1+15),B:
   PUT(X1,Y1)-(X1+15,Y1+15),L
31Ø IF L2=Y2 THEN 33Ø
32Ø PUT(X2,L2)-(X2+15,L2+15),B:
   PUT(X2,Y2)-(X2+15,Y2+15),R
33Ø IF PEEK(34Ø)=223 GOSUB 1ØØØ
34Ø IF PEEK(345)=253 GOSUB 15ØØ
35Ø GOTO 23Ø
36Ø CLS:IF S1>S2 THEN PRINT@96,
   "PLAYER 1 WON BY";S1;"POINTS
   TO";S2:GOTO39Ø
37Ø IF S2>S1 THEN PRINT@96,
   "PLAYER 2 WON BY";S2;"POINTS
   TO";S1:GOTO39Ø
38Ø PRINT@96,"THE GAME WAS
   DRAWN";S1;"POINTS EACH"
39Ø A$=INKEY$:GOTO18Ø
1ØØØ IF B1=Ø THEN RETURN
1Ø1Ø PLAY"T12ØO5AGFEDC"
1Ø2Ø FOR N=32 TO 232 STEP 16
1Ø3Ø LINE(N+1,Y1+7)-(N+6,Y1+7),
   PSET
1Ø4Ø LINE(N+1,Y1+7)-(N+6,Y1+7),
   PRESET
1Ø5Ø NEXT
1Ø6Ø IF Y1=Y2 OR Y1+8=Y2 THEN
   S1=S1+1:CIRCLE(1Ø*S1,8),2,5:
   PLAY"T8O1GDBC"
1Ø7Ø CIRCLE(1Ø*B1,2),1,Ø:B1=B1-1
1Ø8Ø RETURN
15ØØ IF B2=Ø THEN RETURN
151Ø PLAY"T12ØO5BAGFEDC"
152Ø FOR N=216 TO 32 STEP -16
153Ø LINE(N+1,Y2+7)-(N+6,Y2+7),PSET
154Ø LINE(N+1,Y2+7)-(N+6,Y2+7),
   PRESET
155Ø NEXT
156Ø IF Y2=Y1 OR Y2+8=Y1 THEN
   S2=S2+1:CIRCLE(255-1Ø*S2,8),
   2,5:PLAY"T8O1GDBC"
157Ø CIRCLE(255-1Ø*B2,2),1,Ø:
   B2=B2-1
158Ø RETURN
```

RUN the program and see a title page, followed by a scene set for a duel between two space-suited beings. Two people can play this game. The keys to control the beings are as follows. For the Spectrum, 1 moves the left being up, Q down and A fires the laser; Ø, P and E do the same to the right being. For the Acorns, the left player uses Q, A and Z; the right player uses @, semicolon (;) and full stop (.). For the Dragon, the left player uses the up and down arrow keys and Z; the right player uses minus (−), @ and /.

The title page and battle scene are set up between Lines 1Ø and 4ØØ (1Ø and 37Ø on the Acorns and 1Ø and 21Ø on the Dragon). Next comes the main loop, which detects the keypresses and branches the program to routines that move the figures and fire lasers. The loop is the section of the code between Lines 42Ø and 56Ø (38Ø and 48Ø on the Acorns and 23Ø and 35Ø on the Dragon). The rest of the program comprises routines for moving, firing and printing messages.

CONTROLLING THE BOARD

The curtain rises on *INPUT*'s Othello game. Program this deceptively simple game of strategy and take on your computer. But beware, it's no pushover

Othello is a strategy game played on an eight by eight square grid—a chess or draughts board can be used. The rules are very simple, and the game deceptively so.

The object is to capture as many of your opponent's pieces as possible. Play simply consists of each player in turn adding a piece to the board, until the board is full. Each player starts with two pieces and tries to capture those belonging to the other by 'surrounding' them. This is done by placing an extra piece at the end of a row so that the opponent is flanked by your pieces. All the opposing pieces between your pieces are then replaced by yours.

The score is simply the number of pieces that belong to each player that are on the board at any one time. The winner is the player who has the greatest number of pieces when the board becomes full.

On this computer verision, you play against the machine, which also displays the board and keeps track of the score.

HINTS AND TIPS

Like any other strategy game, there are various tricks you can use to help you along. If you have never played Othello before you may find the following hints useful.

The corner pieces are extremely valuable as they cannot be retaken once they are captured—the reason for this is that they cannot be surrounded like any other positions on the board. As a result, they can prove vital to success, and it is well worth capturing the corners even if an alternative move may yield a greater score. Any edge pieces which are touching the corner pieces are also untakeable.

Since a piece can link with more than one line—up and down and diagonally—the most obvious move may not be the best, as in the later stages of the game you can often link two or three lines by adding just one piece.

Think ahead. It may be possible to manoeuvre your opponent into creating opportunities for you to capture vital positions by making a seemingly bad move.

THE PROGRAM

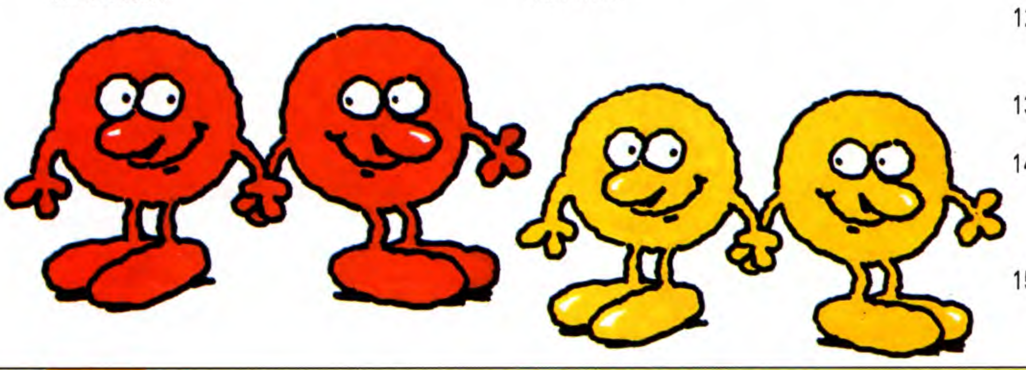
The program takes on the role of your opponent, playing the 'black' pieces, and you'll see that a comparatively simple program can play a quite challenging game of Othello.

One great advantage that a computer program of Othello has over a board game version is that all the hard work is taken away. The computer saves you having to turn pieces over or replace them with ones of a different colour. You are left to concentrate on the game.

Because it considers all the possibilities, the computer takes quite a while to come up with its move, although it speeds up as the game progresses when there are less blank spaces remaining.

PLAYING THE GAME

When you RUN the program you will be asked if you want to go first. When you move you will have to input two coordinates. These make up your position and are in the range one to eight—the row and column numbers are displayed along the top and down one side of the board. The coordinates are entered with the row position first, followed by the column.



The program doesn't recognize a stalemate, nor will it be able to judge if you are bored, so entering Ø as a coordinate will end the game.

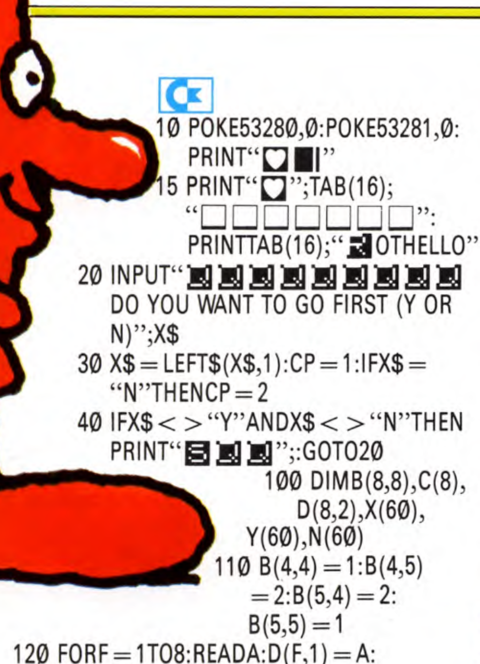
PROLOGUE

Now type in the first section of *INPUT*'s Othello Game. If you RUN the game at this stage you will see the first screen and the graphics for the game, but you won't be able to play the game yet. Don't forget to SAVE the program so that you can add the second section later.

The program for the Acorn computers is rather different from that for the other machines and so follows them, with a separate explanation.



```
10 BORDER Ø: PAPER 7:
   INK 1: CLS
15 PRINT AT 1,11; INVERSE 1;
   "OTHELLO"
20 PRINT AT 1Ø,Ø;"DO YOU WANT TO GO
   FIRST('Y' OR 'N')?": INPUT X$: IF X$=""
   THEN GOTO 2Ø
3Ø LET X$=X$(1): IF X$<>"Y" AND
   X$<>"N" AND X$<>"y" AND
   X$<>"n" THEN GOTO 2Ø
4Ø LET CP=1: IF X$="N" OR X$="n"
   THEN LET CP=2
1ØØ DIM B(8,8): DIM C(8): DIM D(8,2): DIM
   X(6Ø): DIM Y(6Ø): DIM N(6Ø)
11Ø LET B(4,4)=1: LET B(4,5)=2: LET
   B(5,4)=2: LET B(5,5)=1
12Ø FOR F=1 TO 8: READ A:
   LET D(F,1)=A: READ A:
   LET D(F,2)=A: NEXT F
13Ø DATA -1,-1,Ø,-1,1,-1,-1,
   Ø,1,Ø,-1,1,Ø,1,1,1
14Ø FOR F=Ø TO 7: READ A,B,C:
   POKE USR "A"+F,A:
   POKE USR "B"+F,B: POKE USR
   "C"+F,C: NEXT F
15Ø DATA 2Ø4,Ø,Ø,51,6Ø,6Ø,2Ø4,126,
   66,51,126,66,2Ø4,126,66,51,126,66,
   2Ø4,6Ø,6Ø,51,Ø,Ø
```



```
1Ø POKE5328Ø,Ø:POKE53281,Ø:
   PRINT"▢▮"
15 PRINT"▢";TAB(16);
   "▢▢▢▢▢▢▢▢":
   PRINTTAB(16);"▮OTHELLO"
2Ø INPUT"▮▮▮▮▮▮▮▮▮
   DO YOU WANT TO GO FIRST (Y OR
   N)";X$
3Ø X$=LEFT$(X$,1):CP=1:IFX$=
   "N"THENCP=2
4Ø IFX$<>"Y"ANDX$<>"N"THEN
   PRINT"▮▮▮";:GOTO2Ø
1ØØ DIMB(8,8),C(8),
   D(8,2),X(6Ø),
   Y(6Ø),N(6Ø)
11Ø B(4,4)=1:B(4,5)
   =2:B(5,4)=2:
   B(5,5)=1
12Ø FORF=1TO8:READA:D(F,1)=A:
   READA:D(F,2)=A:NEXT
13Ø DATA−1,−1,Ø,−1,1,−1,−1,
   Ø,1,Ø,−1,1,Ø,1,1,1
```

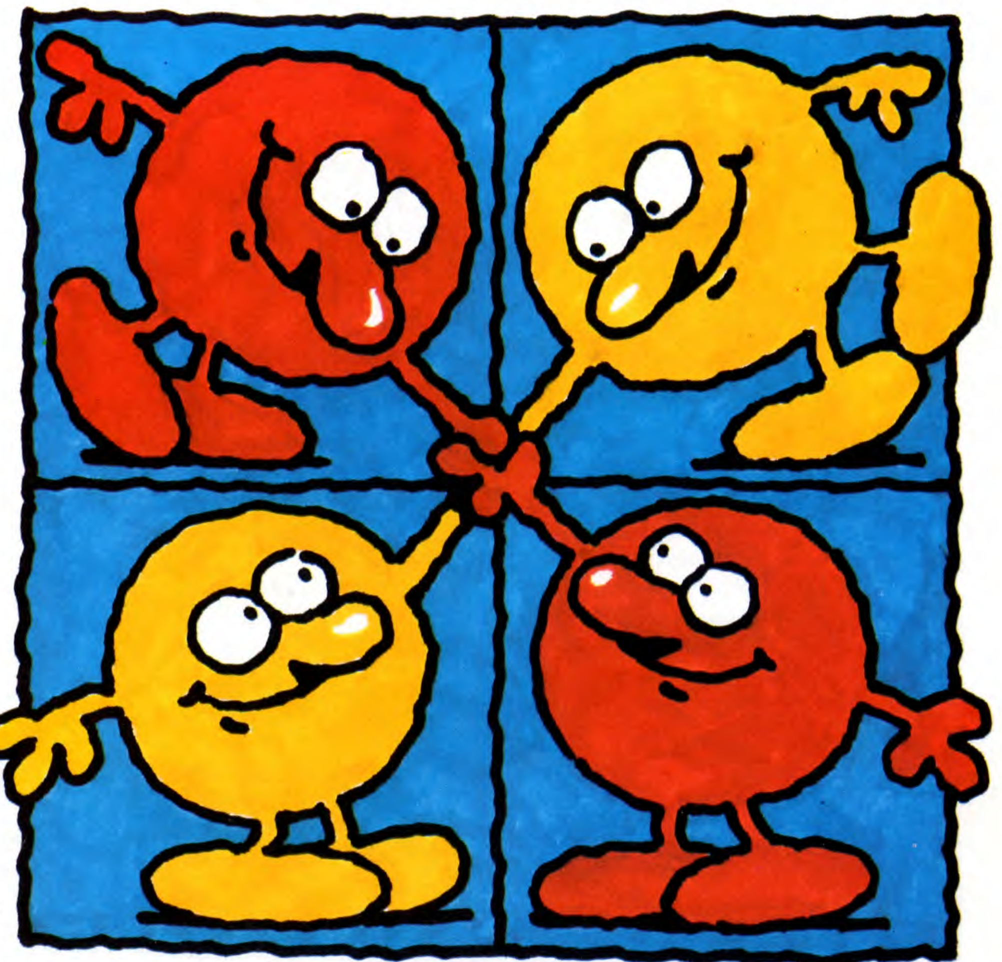
```
1Ø PMODE1,1:COLOR4,1:PCLS:SCREEN
   1,Ø:FOR X=1 TO 16: READ A:
   NEXTX:GOSUB 92ØØ:RESTORE
15 DRAW "BM6Ø,4Ø;S16":A$=
   "OTHELLO":GOSUB 93ØØ
2Ø DRAW "BM26,12Ø;S8;C2":A$=
   "DO YOU WANT TO GO FIRST ":GOSUB
   93ØØ:DRAW "BM1ØØ,14Ø":A$="Y OR
   N":GOSUB 93ØØ
21 X$=INKEY$:IF X$="" THEN 21
3Ø IF X$<>"Y" AND X$
   <>"N" THEN 21
4Ø CP=1:IF X$="N" THEN CP=2
1ØØ DIM B(8,8),C(8),D(8,2),
   X(6Ø),Y(6Ø),N(6Ø)
11Ø B(4,4)=1:B(4,5)=2: B(5,4)=2:
   B(5,5)=1
12Ø FOR F=1 TO 8:READ A:D(F,1)
   =A:READ A:D(F,2)=A:NEXT F
13Ø DATA −1,−1,Ø,−1,1,−1,−1,
   Ø,1,Ø,−1,1,Ø,1,1,1
15Ø GOSUB 11ØØ
```

Lines 1Ø to 4Ø look after the first screen the player sees. Line 1Ø sets the display colours and clears the screen. The game title is displayed by Line 15. In the Dragon/Tandy programs, the high resolution screen has been used, so there's the letter drawing subroutine you have used before at the end of the program. Line 2Ø PRINTs a prompt and asks you if you want to go first. The reply is held in X$. Subsequently, in Line 3Ø, X$ is reduced to its first letter. CP is the current player, and takes the value one for the human and two for the computer. Line 4Ø checks for Y or N.

Lines 1ØØ to 13Ø initialize the variables and arrays needed in the game. Line 1ØØ looks after the arrays. B(x,y) represents the board, and the values stored in each element represent the status of the corresponding square on the board—if an element is zero, the square is empty; if it is one, a piece belonging to the player occupies it; and if the value is two, the square is occupied by the computer's piece. C(x) is used in checking the player's move, D(x,y) contains X and Y displacements for the eight possible directions of movement. X(x), Y(x) and N(x) are all used in calculating the computer's move.

Line 11Ø sets the initial positions on the board. Each player has two pieces positioned at the centre of the board. The possible directions from this position are READ from the DATA in Line 13Ø. Each number represents an X or Y displacement, with the negative numbers being to the left or top of the board.

In the Spectrum version only, Lines 14Ø and 15Ø set up the UDGs for the blank squares, and the two different pieces. Line



14Ø READs the DATA from Line 15Ø and sets up UDGs A, B and C.

ACT 1: THE MAIN LOOP



```
500 GOSUB 1000
505 IF CS+PS=64 THEN GOTO 4000
510 LET EG=0: IF CP=1 THEN GOSUB
    2000: GOSUB 1000: IF EG=1 THEN
    GOTO 4000
515 IF CS+PS=64 THEN GOTO 4000
520 IF CP=2 THEN GOSUB 3000
530 GOTO 500
```



```
500 GOSUB1000
505 IFCS+PS=64THEN4000
510 EG=0:IFCP=1THENGOSUB2000:
    GOSUB1000:IFEG=1THEN4000
515 IFCS+PS=64THEN4000
520 IFCP=2THENGOSUB3000
530 GOTO500
```



```
500 GOSUB 1000
505 IF CS+PS=64 THEN 4000
510 EG=0: IF CP=1 THEN GOSUB
    2000:GOSUB 1000:COLOR1:LINE
    (118,150)-(150,180),PSET,BF:
    IF EG=1 THEN 4000
515 IF CS+PS=64 THEN 4000
520 IF CP=2 THEN GOSUB 3000
530 GOTO 500
```

The main program loop runs from Line 5ØØ to Line 53Ø. Line 5ØØ calls the subroutine which draws the board. Lines 5Ø5 and 515 check to see if the board is full. It does this by checking if CS (the computer score) and PS (the human's score) add up to 64.

Line 51Ø sets the endgame flag (EG) to zero. The middle section of the line calls the player's turn subroutine, followed by the board display subroutine. If, after these subroutines have been executed, EG is set, the program jumps to the endgame routine, starting at Line 4ØØØ. The computer's move is dealt with by Line 52Ø, if CP is 2.

ACT 2: DRAWING THE BOARD



```
1000 CLS : PRINT TAB 11;"12345678": LET
     PS=0: LET CS=0
1010 FOR F=1 TO 8: PRINT TAB 9;F;"□";:
     FOR G=1 TO 8
1020 IF B(F,G)=0 THEN PRINT
     CHR$ 144;
1030 IF B(F,G)=1 THEN PRINT INK 2;
     CHR$ 145;: LET PS=PS+1
1040 IF B(F,G)=2 THEN PRINT INK 2;
     CHR$146;: LET CS=CS+1
1050 NEXT G: PRINT : NEXT F
1052 LET P$="POINTS": IF PS=1 THEN
     LET P$="POINT"
1054 LET Q$="POINTS": IF CS=1 THEN
     LET Q$="POINT"
1060 PRINT " INK 2;"PLAYER=";
     TAB 22;"COMPUTER=": PRINT PS;
     "□";P$;TAB 22;CS;"□";Q$
1070 RETURN
```



```
1000 PRINT"■"TAB(15)"12345
     678■":PS=0:CS=0
1010 FORF=1TO8:PRINTTAB(12);F;:
     FORG=1TO8
1020 IFB(F,G)=0THENPRINT"■■□■";
1030 IFB(F,G)=1THENPRINT"▓●";:
     PS=PS+1
1040 IFB(F,G)=2THENPRINT"■○";:
     CS=CS+1
1050 NEXTG:PRINT"■":NEXTF
1052 P1$="POINTS":IFPS=1THEN
     P1$="POINT"
1054 P2$="POINTS":IFCS=1THEN
     P2$="POINT"
1060 PRINT"■■PLAYER=▓●■"TAB
     (30)"COMPUTER=■○■":PRINTPS;
     P1$;TAB(30)CS;P2$
1070 RETURN
```



```
1000 PS=0:CS=0
1010 FOR F=1 TO 8:FOR G=1 TO 8
1030 IF B(F,G)=1 THEN PAINT
     (G*16+54,F*16),1,3:PS=PS+1
1040 IF B(F,G)=2 THEN PAINT
     (G*16+54,F*16),4,3:CS=CS+1
1050 NEXT G,F
1060 DRAW "BM0,150;C2":A$=
     "PLAYER":GOSUB 9300:DRAW
     "BM186,150":A$="COMPUT":
     GOSUB 9300
1063 LINE (62,150)-(70,158),PSET,
     B:LINE (246,150)-(256,158),
     PSET,B
1064 PAINT(248,152),4,2
1065 COLOR1:LINE (0,60)-(32,80),
     PSET,BF:LINE (216,60)-(248,80),
     PSET,BF
1066 DRAW "BM0,60;S16;C4":A$=MID$
     (STR$(PS),2):GOSUB 9300:DRAW
     "BM216,60":A$=MID$(STR$(CS),
     2):GOSUB 9300:DRAW "S8"
1070 RETURN
1100 PCLS1:COLOR2:FOR X=0 TO 7:
     DRAW "BM"+STR$(X*16+70)+",
     0;S8"+NU$(X+1):NEXTX
1110 LINE(64,12)-(192,140),PSET,
     BF:COLOR 3
1120 FOR X=0 TO 8:LINE (X*16+64,
     12)-(X*16+64,140),PSET:NEXT
1130 FOR Y=0 TO 8:LINE (64,Y*16+
     12)-(192,Y*16+12),PSET:NEXT
1140 FOR Y=0 TO 7:DRAW "C2S8
     BM54,"+STR$(Y*16+14)+NU$
     (Y+1):NEXT
1150 RETURN
```

The board is drawn by the routine which starts at Line 1ØØØ. The screen is cleared, the column numbers are printed, and the two scores set to zero.

The FOR ... NEXT loop between Lines 1Ø1Ø and 1Ø5Ø PRINTs the board on screen, row by row. As the pieces are displayed, the scores are incremented. Lines 1Ø52 and 1Ø54 tidy up the points display, PRINTed in Line 1Ø6Ø.

Lines 11ØØ to 115Ø in the Dragon/Tandy are extra high resolution graphics commands to draw the board.

ACT 3: THE PLAYER'S MOVE



```
2000 PRINT AT 14,0;"WHAT IS YOUR MOVE
     (ROW,COL)?": INPUT X,Y
2005 IF X=0THEN LET EG =1: RETURN
2006 IFX=9 THEN LET CP=2: RETURN
2010 IF X<1 OR X>8 OR Y<1 OR Y>8
     THEN GOTO 2000
2020 IF B(X,Y)=0 THEN GOTO 2070
2040 PRINT AT 17,0;"YOU CANNOT MOVE
     ONTO AN OCCUPIEDSQUARE": FOR F=1
     TO 500: NEXT F
2050 PRINT AT 17,0;"□□□□
     □□□□□□□□□□□□□□
     □□□□□□□□□□□□□□
     □□□□□□□□□□□□":GOTO 2000
2070 LET NF=0: FOR F=1 TO 8: LET
     CF=0: IF X+D(F,1)=0 OR
     X+D(F,1)=9 THEN GOTO 2075
2071 IF Y+D(F,2)=0 OR Y+D(F,2)=9
     THEN GOTO 2075
2072 IF B(X+D(F,1),Y+D(F,2))=2 THEN
     LET CF=1: LET NF=1
2075 LET C(F)=0: IF CF=1 THEN LET
     C(F)=F
2080 NEXT F
2090 STOP
```



```
2000 INPUT"■WHAT IS YOUR MOVE
     (ROW,COL)";X,Y
2005 IFX=0THENEG=1:RETURN
2006 IF X=9 THEN CP=2: RETURN
2010 IF (X<1ORX>8)OR(Y<1ORY>8)
     THEN PRINT"■■■":GOTO2000
2020 IFB(X,Y)=0THEN2070
2040 PRINT"■YOU CANNOT MOVE
```

```
     ONTO AN OCCUPIED SQUARE":
     FORF=0TO1500:NEXT
2050 PRINT"□□□□□□□□□□
     □□□□□□□□□□□□□□
     □□□□□□□□□□□□□□
     □□□□□□□□"
2060 PRINT"□□□□":GOTO2000
2070 NF=0:FORF=1TO8:CF=0:IFX+D
     (F,1)=9ORX+D(F,1)=0THEN2075
2071 IFY+D(F,2)=9ORY+D(F,2)=0
     THEN2075
2072 IFB(X+D(F,1),Y+D(F,2))=2
     THENCF=1:NF=1
2075 C(F)=0:IFCF=1THENC(F)=F
2080 NEXTF
2090 STOP
```

```
2000 COLOR1:LINE (0,182)-(255,191),
     PSET,BF:LINE (118,150)-(150,180),
     PSET,BF:DRAW"C3;BM0,182":A$=
     "ENTER YOUR MOVE ROW AND COL":
     GOSUB 9300:SOUND 100,1
2001 I$=INKEY$:IF I$<"0" OR I$>"9"
     THEN 2001
2002 X=VAL(I$):DRAW "BM118,150;
     S16;C4"+NU$(X)
2003 I$=INKEY$:IF I$<"0" OR I$>"9"
     THEN 2003
2004 Y=VAL(I$):DRAW NU$(Y)+"S8"
2005 IF X=0 THEN EG=1:RETURN
2006 IF X=9 THEN CP=2:RETURN
2010 IF X<1 OR X>8 OR Y<1 OR Y>8
     THEN 2000
2020 IF B(X,Y)=0 THEN 2070
2040 COLOR1:LINE (0,182)-(255,191),
     PSET,BF:DRAW "S8C4BM0,182":A$=
     "YOU CANNOT GO TO THAT SQUARE":
     GOSUB 9300:FOR F=1 TO 900:NEXTF
2050 GOTO 2000
2070 NF=0:FOR F=1 TO 8:CF=0:IF
     X+D(F,1)=0 OR X+D(F,1)=9 THEN
     2075
2071 IF Y+D(F,2)=0 OR Y+D(F,2)=9
     THEN 2075
2072 IF B(X+D(F,1),Y+D(F,2))=2 THEN
     CF=1:NF=1
2075 C(F)=0:IF CF=1 THEN C(F)=F
2080 NEXTF
```

The player's move is input in Lines 2000 to 2270, but this time, the program reaches only Line 2090.

After a prompt, the coordinates (X and Y) are input in Line 2000. Line 2005 checks to see if X is equal to zero, and sets the EG flag.

The player's input is error-checked in Line 2010 and jumps back to Line 2000 if necessary. The chosen square is checked to ensure it is empty, and if it is, the program jumps to Line 2070. If the square is already occupied, Line 2040 PRINTs an error message.

The final pair of lines—Lines 2060 and 2070—in this part of the program check if the new piece has been placed next to one of the computer's pieces.



LETTER-DRAWING ROUTINE

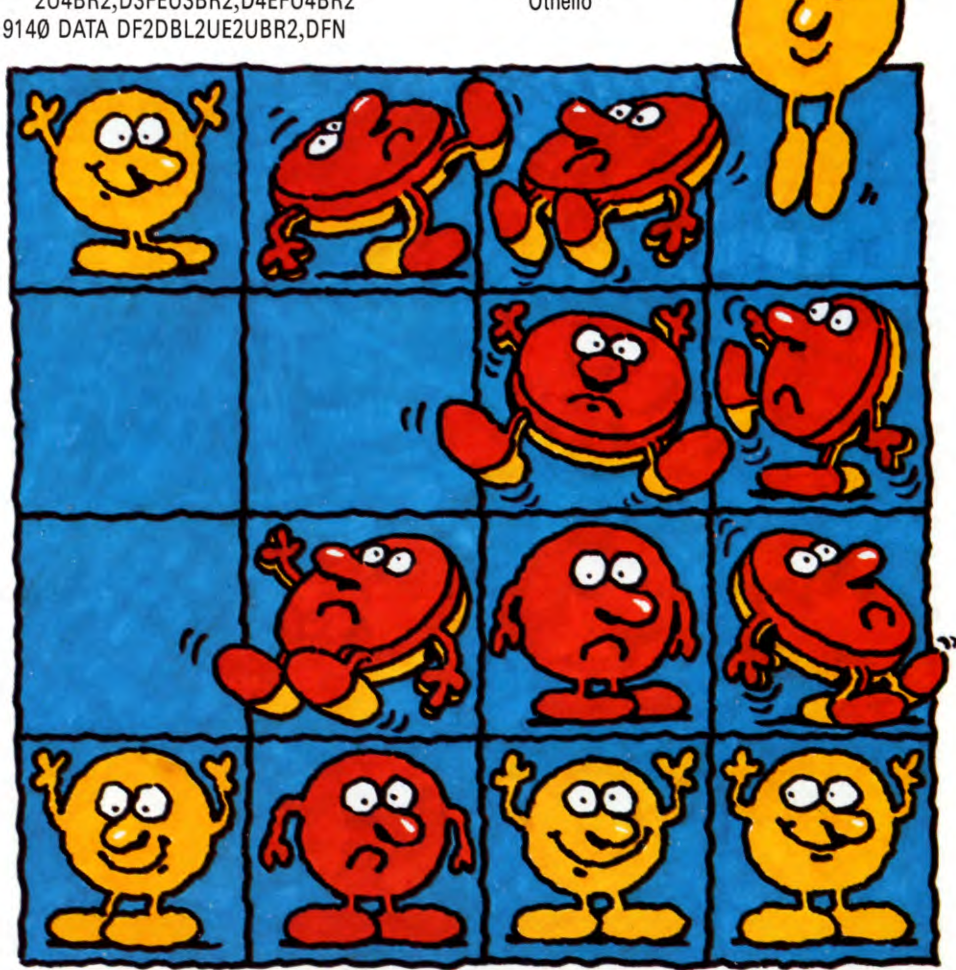
You will need to add this routine to the program to enable you to write on the high-resolution screen. It is similar to the one described in detail on page 192:

```
9100 DATA BR4,ND4R3D2NL3ND2BE2,
     ND4R3DGNL2FDNL3BU4BR2,NR3D4
     R3BU4BR2,ND4R2FD2GL2BE4BR,
     NR3D2NR2D2R3BU4BR2
9110 DATA NR3D2NR2D2BE4BR,NR3
     D4R3U2LBE2BR,D4BR3U2NL3U2BR2,
     ND4BR2,BD4REU3L2R3BR2,D2ND2
     NF2E2BR2
9120 DATA D4R3BU4BR2,ND4FREND
     4BR2,ND4F3DU4BR2,NR3D4R3U4BR2,
     ND4R3D2NL3BE2,NR3D4R3NHU4BR2
9130 DATA ND4R3D2L2F2BU4BR2,BD
     4R3U2L3U2R3BR2,RND4RBR2,D4R
     2U4BR2,D3FEU3BR2,D4EFU4BR2
9140 DATA DF2DBL2UE2UBR2,DFN
     D2EUBR2,R3G3DR3BU4BR2
9150 DATA NR2D4R2U4BR2,BDEND
     4BR2,R2D2L2D2R2BU4BR2,NR2BD
     2NR2BD2R2U4BR2,D2R2D2U4BR2,
     NR2D2R2D2L2BE4,D4R2U2L2BE2
     BR2,R2ND4BR2,NR2D4R2U2NL2U
     2BR2,NR2D2R2D2U4BR2
9200 DIM LE$(26)
9210 FOR K=0 TO 26:READ LE$(K):NEXT
9220 FOR K=0 TO 9:READ NU$(K):NEXT
9230 RETURN
9300 FOR K=1 TO LEN(A$)
9310 B$=MID$(A$,K,1)
9320 IF B$>="0" AND B$<="9" THEN
     DRAW NU$(VAL(B$)):GOTO 9350
9330 IF B$="□" THEN N=0 ELSE
     N=ASC(B$)-64
9340 DRAW LE$(N)
9350 NEXT
9360 RETURN
```



PROLOGUE

```
10 MODE6:VDU
   19,0,4;0;0;
20 PRINT TAB(15,5);
   "Othello"
```



```
30 PRINT TAB(4,10);"Do you
   want to go first (Y/N) ?";
40 REPEAT
50 x$=GET$
60 UNTIL INSTR("YyNn",x$)>0
70 PRINT x$
80 cp%=1:IF x$="N" OR x$="n" THEN
   cp%=2
90 DIM b%(8,8),c%(8),d%(8,2),
   x%(60),y%(60),n%(60)
100 b%(4,4)=1: b%(4,5)=2
110 b%(5,4)=2: b%(5,5)=1
120 FOR f%=1 TO 8:READ d%(f%,1): READ
   d%(f%,2): NEXT
130 DATA -1,-1,0,-1,1,-1,-1,0,1,
   0,-1,1,0,1,1,1
140 VDU 23,241,0,60,126,126,126,
   126,60,0
150 MODE5
160 VDU 19,2,4,0;0;
170 VDU 23;8202;0;0;0;
```

Lines 10 to 80 look after the initial screen. The screen colours and mode are set up, then the title and the first go prompt are displayed. The choice is error-checked in Line 60.

Lines 90 to 170 initialize the display. b%(x,y) represents the board, and the values stored in each element represent the status of the corresponding square on the board. If an element is zero, the square is empty; if it is one, a piece belonging to the player occupies it; and if it is two, the square is occupied by one of the computer's pieces. c%(x) is used in checking the player's move, d%(x,y) contains x and y displacements for the eight possible directions of movement.

Lines 100 and 110 set the initial positions of the pieces on the board. Each player has two pieces positioned at the centre of the board. The DATA in Line 130 are the possible displacements from this initial position, with the negative numbers being to the left or top of the board.

THE MAIN LOOP

```
180 REPEAT
190 PROCdisplayboard
200 IF (cs%+ps%)=64 THEN PROCgameover
210 LET eg%=0:IF cp%=1 THEN
   PROChumanmove:IF eg%=1 THEN
   PROCgameover
220 IF (cs%+ps%)=64 THEN PROCgameover
230 IF cp%=2 THEN PROCcomputermove
240 UNTIL FALSE
```

This is the core of the program. From these few lines, the remainder of the program is called. Line 190 displays the board, and if the board is full, Line 200 or Line 220 call PROCgame over.

Line 210 checks the end game flag (eg%), and cp% for the human's move. If cp% is two, then Line 230 calls PROCcomputer move.

DRAWING THE BOARD

```
250 DEFPROCdisplayboard
260 CLS
270 COLOUR 3:PRINT TAB(6,6);
   "12345678"
280 LET ps%=0:LET cs%=0
290 FOR f%=1 TO 8
300 COLOUR 128:COLOUR 3:PRINT
   TAB(4,f%+7);f%
310 FOR g%=1 TO 8
320 COLOUR 129:COLOUR 0
330 IF (g%+f%) MOD 2=0 THEN COLOUR
   130
340 IF b%(f%,g%)=0 THEN PRINT
   TAB(f%+5,7+g%);"□";
350 IF b%(f%,g%)=1 THEN PRINT
   TAB(f%+5,7+g%);CHR$(241);:
   ps%=ps%+1:SOUND 1,-8,RND(50)
   +50,1
360 IF b%(f%,g%)=2 THEN COLOUR
   3:PRINT TAB(f%+5,7+g%);
   CHR$(241);:cs%=cs%+1
370 NEXT g%:PRINT:NEXT f%
380 p$="□Points□□":IF ps%=1 THEN
   p$="□Point□□□"
390 q$="□Points□□":IF cs%=1 THEN
   LET q$="□Point□□□"
400 COLOUR 128:COLOUR 3
410 PRINT TAB(2,1);"You□:";
   ps%;p$
420 PRINT TAB(2,20);"Me□□:";
   cs%;q$
430 ENDPROC
```

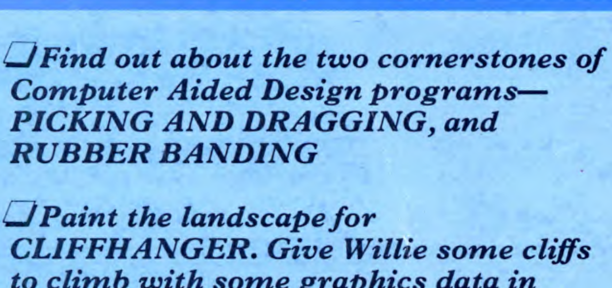


PROCdisplay board redraws the board column by column after each move, plotting the pieces in the correct colours. In addition, the scores for both the player and the computer are calculated and displayed at the side of the board.

CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

COMING IN ISSUE 32...



- Find out about the two cornerstones of Computer Aided Design programs—PICKING AND DRAGGING, and RUBBER BANDING

- Paint the landscape for CLIFFHANGER. Give Willie some cliffs to climb with some graphics data in part four of the machine code game

- Use the CALENDAR PROGRAM to remind yourself of all those things you normally forget—birthdays, TV licence, you name it!

- Complete the OTHELLO game and issue a challenge to your computer—it's now ready to take you on at this game of strategy

- Take your musical skills one stage further. Learn how to play CHORDS on the COMMODORE 64 and BBC

A MARSHALL CAVENDISH 32 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE

ASK YOUR NEWSAGENT FOR INPUT